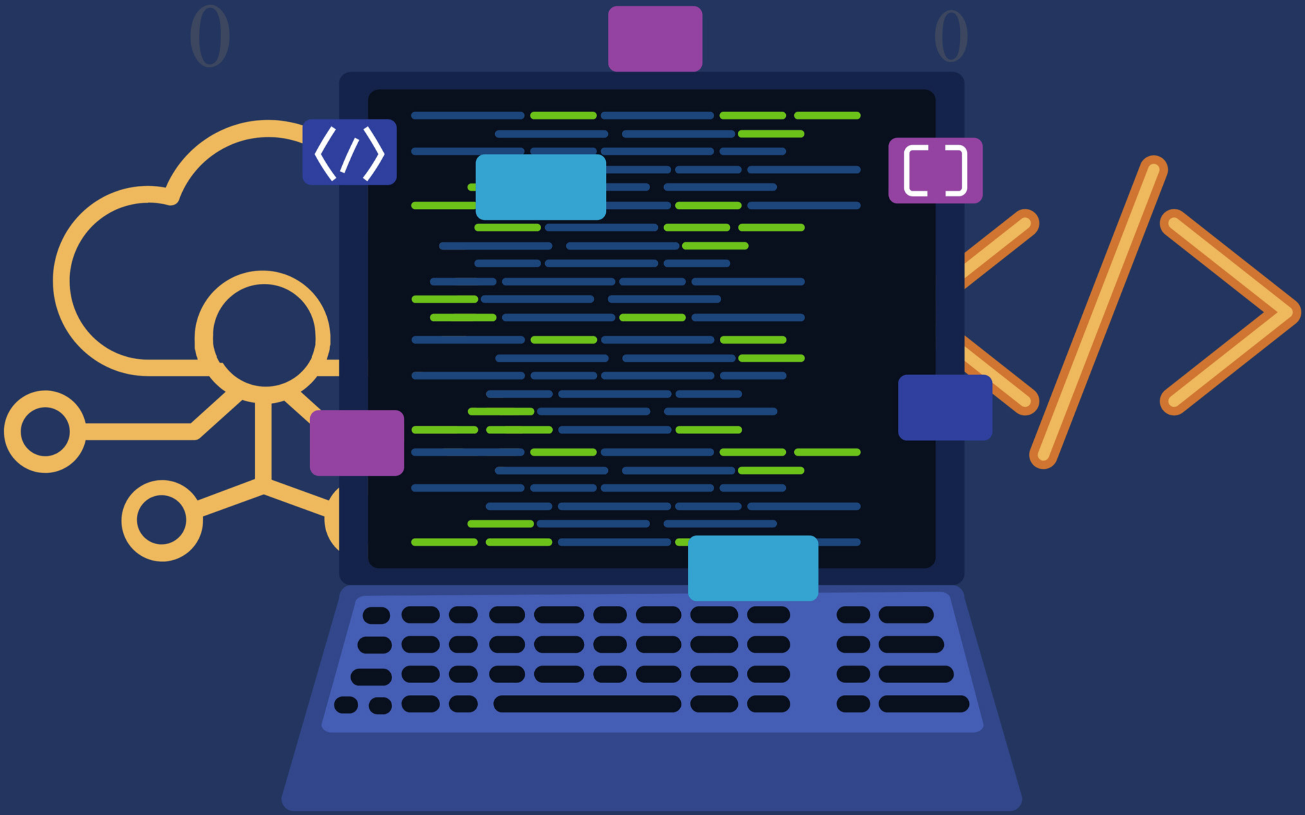
Introduction to Python Programming

Course Code: B24DS06DC
BSc Data Science and Analytics
Discipline Core Course
Self Learning Material



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Introduction to Python Programming

Course Code: B24DS06DC

Semester - III

Discipline Core Course
Undergraduate Programme
BSc Data Science and Analytics
Self Learning Material
(With Model Question Paper Sets)

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

INTRODUCTION TO PYTHON PROGRAMMING

Course Code: B24DS06DC

Semester- III

Discipline Core Course

BSc Data Science and Analytics

SREENARAYANAGURU OPEN UNIVERSITY

Scan this QR Code for reading the SLM on a digital device.

Edition
October 2025

MESSAGE FROM VICE CHANCELLOR

Dear learner,

I extend my heartfelt greetings and profound enthusiasm as I warmly welcome you to Sreenarayanaguru Open University. Established in September 2020 as a state-led endeavour to promote higher education through open and distance learning modes, our institution was shaped by the guiding principle that access and quality are the cornerstones of equity. We have firmly resolved to uphold the highest standards of education, setting the benchmark and charting the course.

The courses offered by the Sreenarayanaguru Open University aim to strike a quality balance, ensuring students are equipped for both personal growth and professional excellence. The University embraces the widely acclaimed "blended format," a practical framework that harmoniously integrates Self-Learning Materials, Classroom Counseling, and Virtual modes, fostering a dynamic and enriching experience for both learners and instructors.

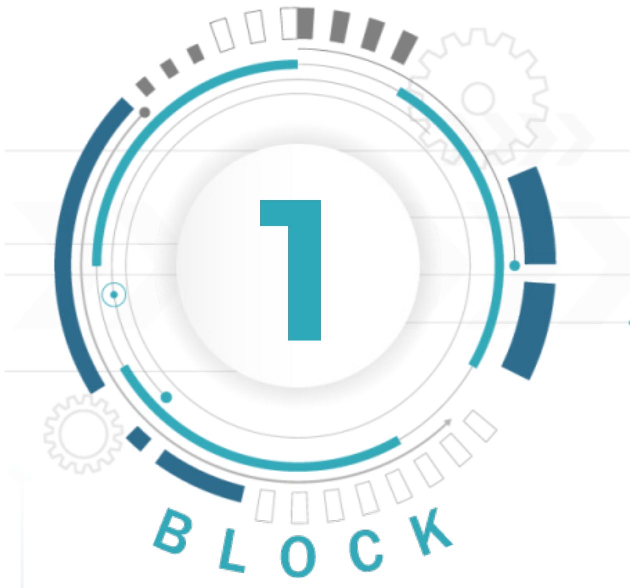
The University is committed to providing an engaging and dynamic educational environment that encourages active learning. The Study and Learning Material (SLM) is specifically designed to offer you a comprehensive and integrated learning experience, fostering a strong interest in exploring advancements in information technology (IT). The curriculum has been carefully structured to ensure a logical progression of topics, allowing you to develop a clear understanding of the evolution of the discipline. It is thoughtfully curated to equip you with the knowledge and skills to navigate current trends in IT, while fostering critical thinking and analytical capabilities.The Self-Learning Material has been meticulously crafted, incorporating relevant examples to facilitate better comprehension.

Rest assured, the university's student support services will be at your disposal throughout your academic journey, readily available to address any concerns or grievances you may encounter. We encourage you to reach out to us freely regarding any matter about your academic programme. It is our sincere wish that you achieve the utmost success.

[illegible]

Regards,
Dr. Jagathy Raj V. P. 01-10-2025

Contents



Python Fundamentals

Unit 1

Introduction to Python and Setup

Learning Outcomes

After completing this unit, the learner will able to:

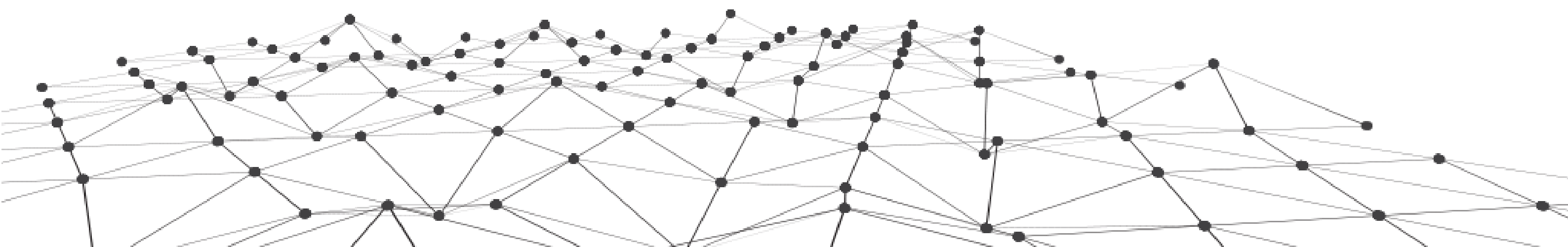
- define Python as a high-level programming language.
- demonstrate the use of Python in interactive mode to execute basic commands.
- navigate and use a Python IDE to write and run Python scripts.
- define what a variable is and apply Python's rules for naming variables correctly.
- compare Python's features with other high-level languages to understand its advantages.

Prerequisites

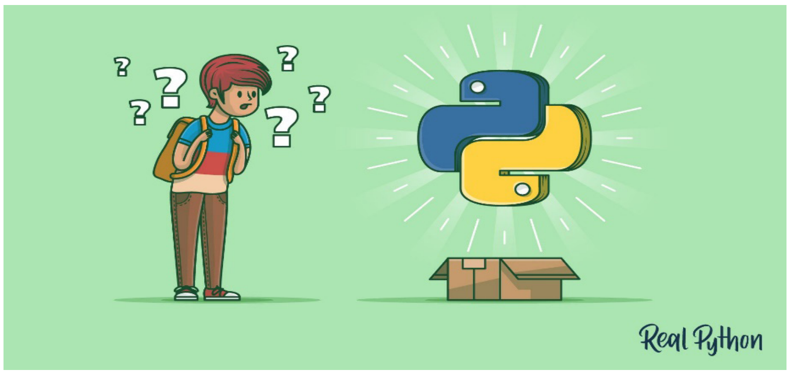
Have you ever wondered how your favourite apps like Instagram, YouTube, or even your smart assistant work behind the scenes? These applications are built using programming languages and one of the most popular and beginner-friendly languages used today is Python. Python is not just used by tech giants like Google and Netflix, but also by scientists, engineers, and students to build programs that solve real-world problems. Before diving in, all you need is basic computer skills, such as opening files, using a web browser, and typing comfortably.

Programming languages can be compared to human languages they help us communicate instructions to computers. High-level languages like Python are designed to be easy to understand, with simple words like `print` and `input` that resemble English. For example, a line of Python code can tell a computer: *"Print a welcome message on the screen,"* just like writing a sentence. This unit will guide you through using Python in interactive mode, where you can type a command and see the result instantly like asking a calculator to solve a math problem in real time.

To make writing and managing code even easier, we use tools called IDEs (Integrated Development Environments) like IDLE or VS Code that help organize your work, highlight mistakes, and make coding more efficient. You'll also learn how to name and



use variables, which are like labelled containers for storing information. For example, in a shopping app, a variable called `price` could store the cost of an item. By the end of this unit, you'll be equipped to write basic Python programs and understand how professional developers use these same tools to build software that you use every day.



Key words

Programming Language, Installation, Python Interactive Mode, Python IDE, Jupyter Notebook, PyCharm

Discussion

1.1.1 Introduction to Program

WhatsApp is an application for communication. A team of people who created the application includes programmers. They have written the program to do various tasks using the application. Imagine the different tasks that can be done using WhatsApp. (Hint: one of the tasks is to send the message). To send a message we need to:

- Open the application
- Select the sender's contact
- Type message
- Send the message

These four steps form the task to send a message using WhatsApp. To carry out a task by computer, we need to give instructions to the computer. A program is a set of instructions that makes a computer usable. The tasks are written using a programming language. Nowadays, we use many applications on mobile phones, computers, and many other devices. Have you thought about the way the application functions and it is made? An application is a program created to perform specific tasks by an end-user.

The set of instructions to carry out a task is called a program. The instructions can be written by using a programming language. The instructions should be understandable by the device. Python is one of the programming languages to write instructions. We are using our mother tongue for communication. It is a means of giving instructions,

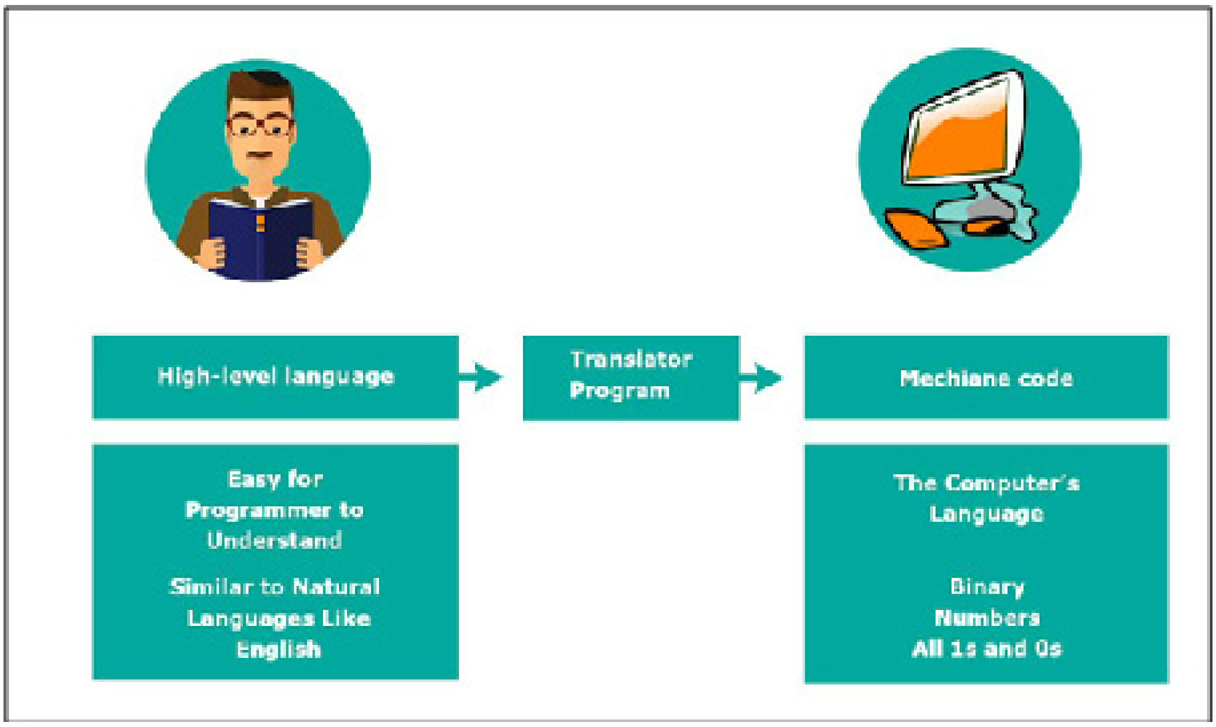

High-level language
Translator Program
Mechiane code
Easy for Programmer to Understand
Similar to Natural Languages Like English
The Computer's Language
Binary Numbers All 1s and 0s

Fig 1.1.1 Translation of high level language to machine language

expressing our feelings, etc. Similarly, programming languages are used to make computer programs or software applications. A language is made up of alphabets, words, syntax, and semantics. The syntax is a set of rules to make a valid sentence. Semantic is a set of rules that determine the meaning of the sentences. A person who has the skill and knowledge of programming is a programmer.

1.1.2 Python as High-level Language

High-Level Language is a programming language easily understandable by humans. Python is a high-level language. Other high-level languages include Java, C++, PHP, Ruby, Perl, etc. The programs written in a high-level language need to be translated into machine language. Machine Language is a low-level language that machines understand. Machine code is a sequence of numbers written in binary (0 and 1). The program that translates high-level language to machine code is called a compiler or an interpreter. Fig 1.1.1 shows translation of high level language to machine language.

Python was created by Guido Van Rossum and released in 1991. Python is a well-designed programming language. Python is useful for accomplishing real-world tasks. Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

We can use Python for everything from website development and IoT, gaming, robotics, implementing standalone programs, and many more. Python is used widely to implement complex Internet services like search engines, cloud storage and tools, social media, and so on. For example, Google uses Python language to make the search engine better and more efficient. Google's main search algorithms are written in C++ and Python. One of the most popular languages used in Machine learning is Python. The availability of a wide collection of library functions of Python makes the programming easy and effective.

When Python is installed on a computer, it installs several components such as an interpreter and supporting library. The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications. The set of instructions written in a high-level programming language (For example, Python) is the source code and the file is called a source file. Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement.
- statement grouping is done by indentation instead of beginning and ending brackets.
- no variable or argument declarations are necessary.

1.1.3 Advantages and Disadvantages of Python Programming Language

Python is a widely used high-level programming language known for its simplicity, readability, and versatility. It has gained popularity among beginners and professionals alike due to its clean syntax and powerful capabilities. Python is used in various domains such as web development, data analysis, artificial intelligence, machine learning, and automation, making it one of the most in-demand languages in the tech industry.

However, like any programming language, Python comes with its own set of advantages and disadvantages. While it offers many features that enhance productivity and ease of development, it also has certain limitations that may affect performance or suitability for specific tasks. Understanding both the strengths and weaknesses of Python is essential for making informed choices in software development.

Advantages of Python Programming Language

1. **Presence of third-party modules:** Python has a rich ecosystem of third-party modules and libraries that extend its functionality for various tasks.
2. **Extensive support libraries:** Python boasts extensive support libraries like NumPy for numerical calculations and Pandas for data analytics, making it suitable for scientific and data-related applications.
3. **Open source and large active community base:** Python is open source, and it has a large and active community that contributes to its development and provides support.

4. **Versatile, easy to read, learn, and write:** Python is known for its simplicity and readability, making it an excellent choice for both beginners and experienced programmers.

5. **User-friendly data structures:** Python offers intuitive and easy-to-use data structures, simplifying data manipulation and management.

6. **High-level language:** Python is a high-level language that abstracts low-level details, making it more user-friendly.

7. **Dynamically typed language:** Python is dynamically typed, meaning you don't need to declare data types explicitly, making it flexible but still reliable.

8. **Object-Oriented and Procedural programming language:** Python supports both object-oriented and procedural programming, providing versatility in coding styles.

9. **Portable and interactive:** Python is portable across operating systems and interactive, allowing real-time code execution and testing.

10. **Ideal for prototypes:** Python's concise syntax allows developers to prototype applications quickly with less code.

11. **Highly efficient:** Python's clean design provides enhanced process control, and it has excellent text processing capabilities, making it efficient for various applications.

12. **Internet of Things (IoT) opportunities:** Python is used in IoT applications due to its simplicity and versatility.

13. **Interpreted language:** Python is interpreted, which allows for easier debugging and code development.

Disadvantages of Python Programming Language

1. **Performance:** Python is an interpreted language, which means that it can be slower than compiled languages like C or C++. This can be an issue for performance-intensive tasks.

2. **Global Interpreter Lock:** The Global Interpreter Lock (GIL) is a mechanism in Python that prevents multiple threads from executing Python code at once. This can limit the parallelism and concurrency of some applications.

3. **Memory consumption:** Python can consume a lot of memory, especially when working with large datasets or running complex algorithms.

4. **Dynamically typed:** Python is a dynamically typed language, which means that the types of variables can change at runtime. This can make it more difficult to catch errors and can lead to bugs.

5. **Packaging and versioning:** Python has a large number of packages and libraries, which can sometimes lead to versioning issues and package conflicts.

6. **Lack of strictness:** Python's flexibility can sometimes be a double-edged sword. While it can be great for rapid development and prototyping, it can also lead to code that is difficult to read and maintain.

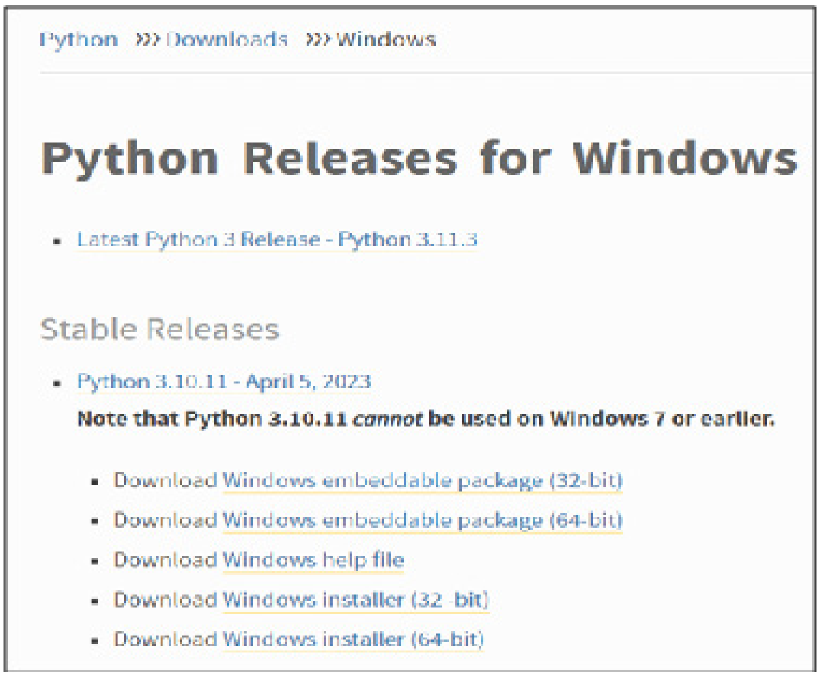
7. **Steep learning curve:** While Python is generally considered to be a relatively easy language to learn, it can still have a steep learning curve for beginners, especially if they have no prior experience with programming.

1.1.4 Python Installation

Python is a powerful and beginner-friendly programming language used for web development, data analysis, artificial intelligence, and more. Before writing and running Python code, the first step is to install Python on your computer. Installing Python sets up the tools needed to write, test, and run programs in an efficient and interactive way. Whether you're using Windows, MacOS, or Linux, Python provides a simple installer and supports various development environments to get you started quickly. This section will guide you through downloading Python from the official website, installing it step-by-step, and verifying the installation so you're ready to begin coding.

1.1.4.1 Installing Python on Windows

We've outlined clear, step-by-step instructions to help you complete the installation process successfully. Whether you're a beginner or have some programming experience, learning how to install Python on Windows is the first step toward harnessing the power of this versatile language and exploring its wide array of uses. To get Python on your computer, follow these steps:



Python >>> Downloads >>> Windows

Python Releases for Windows

- Latest Python 3 Release - Python 3.11.3

Stable Releases

- Python 3.10.11 - April 5, 2023

  Note that Python 3.10.11 *cannot* be used on Windows 7 or earlier.

  - Download Windows embeddable package (32-bit)
  - Download Windows embeddable package (64-bit)
  - Download Windows help file
  - Download Windows installer (32-bit)
  - Download Windows installer (64-bit)

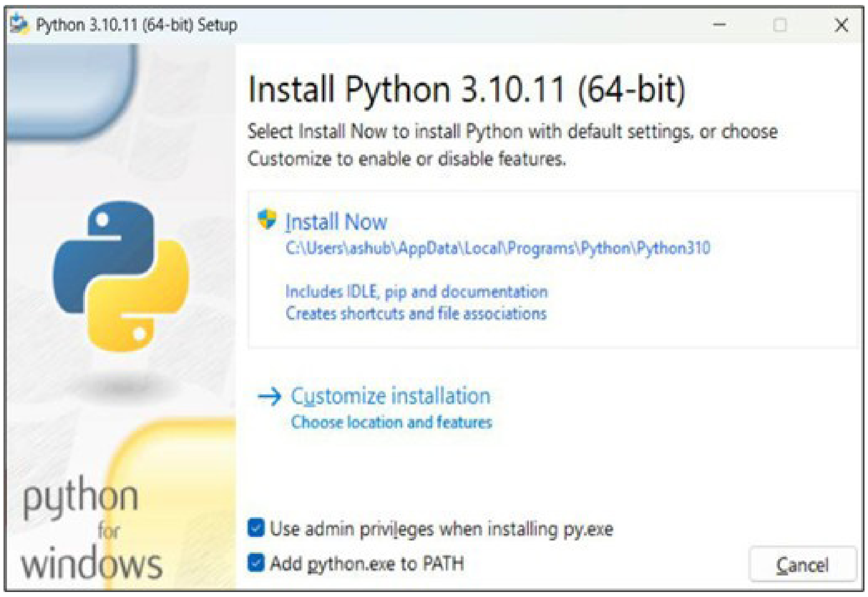
Fig 1.1.2 Python Homepage

Step 1: Choose the Python Version to Install

Go to the official Python website at https://www.python.org/downloads/ using a Windows system. Look for a stable release of Python 3 ideally version 3.10.11, as that's the one used in this section. From the available options, select the appropriate installer for your system: either the 64-bit or 32-bit Windows installer (fig 1.1.2). Then, download the executable file.

Step 2: Downloading the Python Installer

Once you have downloaded the installer, open the .exe file, such as python-3.10.11-amd64.exe, by double-clicking it to launch the Python installer. Choose the option to Install the launcher for all users by checking the corresponding checkbox, so that all users of the computer can access the Python launcher application. Enable users to run Python from the command line by checking the Add python.exe to PATH checkbox.


Python 3.10.11 (64-bit) Setup

Install Python 3.10.11 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\ashub\AppData\Local\Programs\Python\Python310

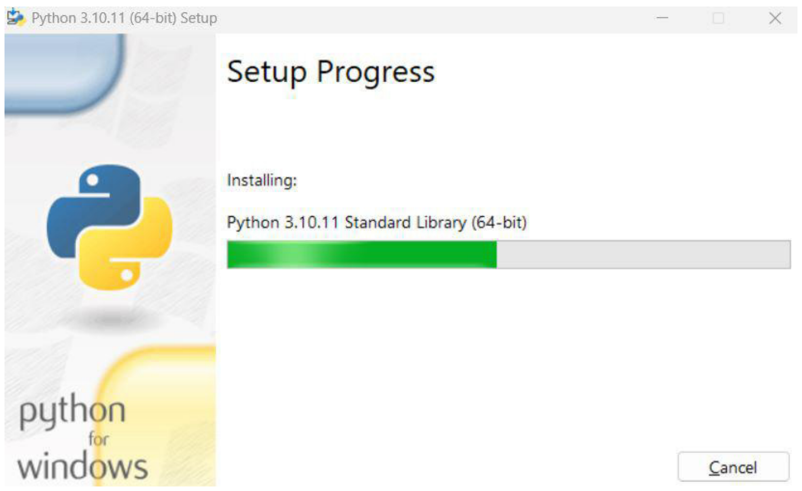
Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Use admin privileges when installing py.exe
☑ Add python.exe to PATH

Fig 1.1.3 Python Installer

After Clicking the **Install Now Button** (fig 1.1.3**)** the setup will start installing Python on your Windows system. You will see a window like this (fig 1.1.4).


Python 3.10.11 (64-bit) Setup

Setup Progress

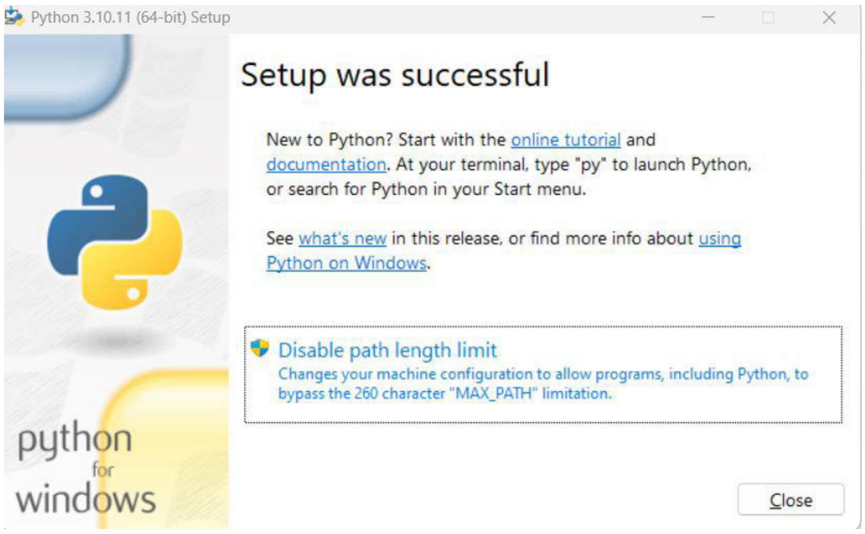
Installing:

Python 3.10.11 Standard Library (64-bit)

Fig 1.1.4 Python Setup

Step 3: Running the Executable Installer

After completing the setup. Python will be installed on your Windows system. You will see a successful message shown in fig 1.1.5.


Python 3.10.11 (64-bit) Setup

Setup was successful

New to Python? Start with the online tutorial and documentation. At your terminal, type "py" to launch Python, or search for Python in your Start menu.

See what's new in this release, or find more info about using Python on Windows.

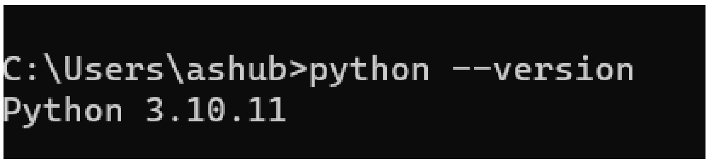
Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.

Fig 1.1.5 Python successfully installed

Step 4: Verify the Python Installation in Windows

Close the window after successful installation of Python. You can check if the installation of Python was successful by using either the command line or the **Integrated Development Environment (IDLE)**, which you may have installed (fig 1.1.6). To access the command line, click on the Start menu and type “cmd” in the search bar. Then click on Command Prompt. Then type as:

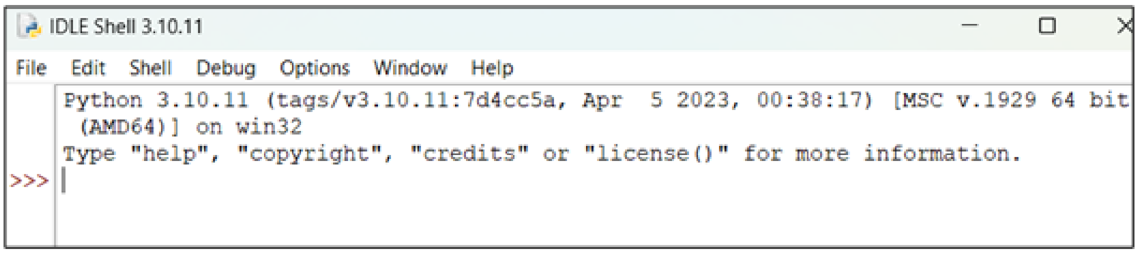
python --version



```
C:\Users\ashub>python --version
Python 3.10.11
```

Fig 1.1.6 Python version

You can verify the installed Python version by opening the IDLE (Integrated Development and Learning Environment) application shown in fig 1.1.7. Simply go to the Start menu, type “IDLE” in the search bar, and click on the IDLE app, for example, **IDLE (Python 3.10.11 64-bit)**. If the IDLE window opens successfully, it means Python has been correctly downloaded and installed on your Windows system.



```
Python 3.10.11 (tags/v3.10.11:7d4cc5a, Apr  5 2023, 00:38:17) [MSC v.1929 64 bit
 (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

Fig 1.1.7 Python IDLE

1.1.4.2 Installing Python on Linux

Installing Python on a Linux system is a straightforward process that allows users to harness the full power of this popular programming language in an open-source environment. Most modern Linux distributions come with Python pre-installed; however, users may need to install or upgrade to a specific version for development or compatibility purposes. This guide will walk you through the steps to check the current Python version, install a new version using package managers like `apt`, `yum`, or `dnf`, and set up tools like `pip` and virtual environments to manage your Python projects efficiently. Whether you're a beginner or an experienced developer, understanding how to install and configure Python on Linux is a key step in mastering Python programming.

Simple Steps to Install Python on Linux are as follows:

Step 1: Check if Python is already installed

Open the terminal and type:

```
python3 --version
      or
python -version
```

If a version number is shown, Python is already installed.

Step 2: Update Package List

Open your terminal and run:

```
sudo apt update
```

This updates the list of available packages.

Step 3: Install Prerequisites

Install necessary software for building Python:

```
sudo apt install -y software-properties-common
sudo apt install -y build-essential libssl-dev zlib1g-dev \
libncurses5-dev libncursesw5-dev libreadline-dev libsqlite3-dev \
libgdbm-dev libdb5.3-dev libbz2-dev libexpat1-dev liblzma-dev \
tk-dev libffi-dev wget
```

Step 4: Download Python Source Code

Go to the official Python website and copy the link for the version you want. Then use `wget` to download:

```
wget https://www.python.org/ftp/python/3.12.2/Python-
3.12.2.tgz
```

Step 5: Extract the Archive (Unpack the downloaded file)

```
tar -xvf Python-3.12.2.tgz
cd Python-3.12.2
```

Step 6: Configure and Install (Run the following commands)

```
./configure --enable-optimizations

make -j$(nproc)

sudo make altinstall
```

Use `make altinstall` instead of `make install` to avoid replacing the system's default Python.

Step 7: Verify the Installation

Check the installed version:

```
python3.12 –version
```

Python is now installed.

1.1.5 Python Interactive Mode

Python offers an *interactive mode* that allows users to type and execute code one line at a time. This mode is useful for beginners to quickly test small pieces of code and see the results immediately. It behaves like a real-time calculator for programming whatever you type is instantly executed, and the output is displayed right away.

Python Interactive Mode is a way of using the Python interpreter directly by typing commands into the Python shell or terminal. It is identified by the **>>>** prompt, which indicates that Python is ready to accept a command. You can launch interactive mode simply by opening a terminal and typing **`python`** or **`python3`** depending on your system setup. Some key features are:

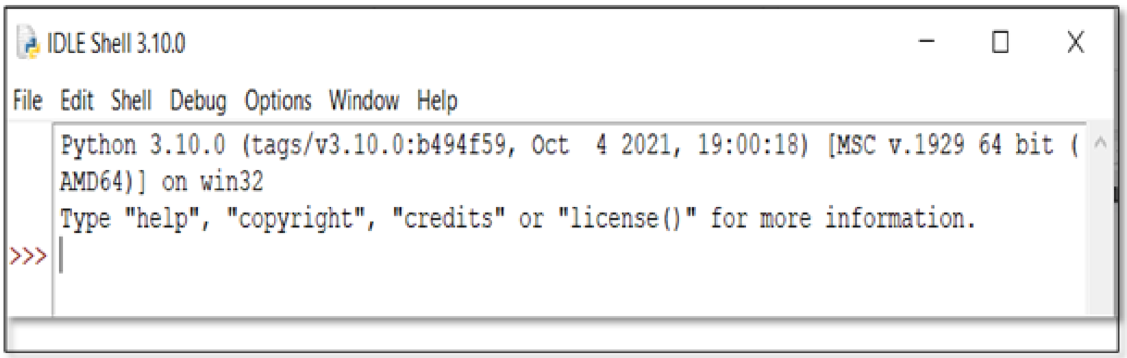
1. **Immediate Execution**: Code is executed line-by-line as you type.
2. **Quick Testing**: Useful for testing small snippets of code or expressions without writing a full program.
3. **Interactive Debugging**: Helps understand how code works by running commands interactively.
4. **Dynamic Typing Display**: Results of expressions are displayed automatically.
5. **Supports Multiline Statements**: Indented blocks, loops, and function definitions can be entered interactively.

How to start Python Interactive Mode?

Step 1: Open a terminal (Command Prompt on Windows, Terminal on macOS/Linux).

Step 2: Type `python` or `python3` (depending on your installation) and press **Enter**.

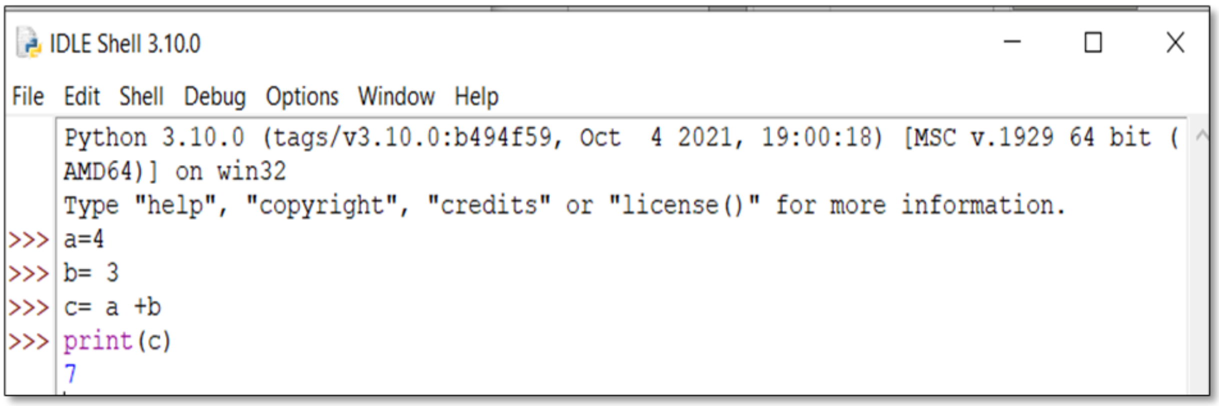
Step 3: You will see the Python prompt (Fig 1.1.8): **>>>**



```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Fig 1.1.8 Python prompt

Step 4: Now, you can enter Python commands (Fig 1.1.9) and see results immediately (Here is an example for addition of two numbers).



```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> a=4
>>> b= 3
>>> c= a +b
>>> print(c)
7
```

Fig 1.1.9 Sample Example

Step 5: Exiting Interactive Mode

Type `exit()` or press `Ctrl + Z` (Windows) / `Ctrl + D` (Linux/macOS), then press Enter.

Activity: Perform simple arithmetic operations using Python **Interactive Mode.**

Python Interactive Mode is a powerful tool for beginners and developers alike to test code snippets and learn Python by immediate execution. It is an essential feature for quick experimentation, debugging, and understanding Python commands.

1.1.6 Python IDE

A **Python Integrated Development Environment (IDE)** is a software application designed to provide programmers with a comprehensive and convenient environment for writing, editing, running, and debugging Python code. Unlike a simple text editor, a Python IDE combines multiple useful features such as syntax highlighting, code completion, error checking, and debugging tools all within a single interface. The primary purpose of a Python IDE is to improve the productivity and efficiency of developers by streamlining the process of software development. It helps programmers by automatically identifying mistakes, suggesting possible code completions, and allowing them to test their code quickly without switching between multiple tools. Python IDEs are especially helpful for beginners learning the language as well as professional developers working on large and complex projects. By providing an organized workspace where code files, libraries, and related resources can be managed easily, Python IDEs also simplify project management and collaboration. Overall, a Python IDE acts as a powerful assistant that supports developers at every stage of the coding lifecycle, from writing clean code to debugging and maintaining software.

Common Features of Python IDEs

Python Integrated Development Environments (IDEs) come equipped with several important features that help programmers write and manage code more efficiently. These features make coding easier, faster, and less error-prone. Some of the most common features include:

Code Editor: The core part of any IDE is the code editor, where programmers write their Python scripts. It usually includes **syntax highlighting**, which displays keywords, variables, and other code elements in different colors to improve readability. Many IDEs also offer **code completion** or **autocomplete** suggestions that help speed up coding by predicting what the programmer intends to type next. Additionally, features like automatic **indentation** and **formatting** help maintain proper code structure, which is important in Python.

1. **Debugger:** Debugging tools allow programmers to find and fix errors in their code. A debugger lets users **step through the program line-by-line, set breakpoints** to pause execution at specific points, and **inspect variables** to see their values at runtime. This makes it easier to understand how the program behaves and identify logical errors or bugs.

2. **Run and Execute Environment:** IDEs provide the ability to **run Python programs directly** from the interface. Output from the program, including errors and print statements, appears within the IDE itself, allowing quick testing and feedback without switching to a separate command-line window.

3. **Project Management:** For larger projects, IDEs help organize multiple files and folders within a **workspace or project explorer.** This makes it easier to navigate through the code-base, manage dependencies, and work on different parts of a project efficiently.

4. **Version Control Integration:** Many Python IDEs include built-in support for **version control systems** like Git. This allows programmers to track changes in their code, collaborate with others, and manage different versions of their projects seamlessly.

5. **Additional Tools:** Other helpful tools in Python IDEs include **code linting** (which checks for syntax errors and coding style issues), **refactoring support** (to improve code structure without changing behavior), and integration with **testing frameworks** for running unit tests.

These features collectively provide a rich and supportive environment that helps programmers write cleaner code, debug efficiently, and manage projects with ease.

1.1.7 Popular Python IDEs

There are many Integrated Development Environments (IDEs) available for Python programming, each designed to cater to different needs and preferences of developers. Some IDEs offer a simple and lightweight interface ideal for beginners, while others provide powerful features suited for professional software development and large projects. Choosing the right Python IDE can greatly enhance coding efficiency, ease of debugging, and overall productivity. Below are some of the most popular and widely used Python IDEs (Table 1.1.1), each with unique strengths that make them favored by learners, data scientists, and professional developers alike.

Table 1.1.1 Popular Python IDEs

IDE Name	Description	Suitable For
Jupyter Notebook	Interactive web-based environment for code, text, and visualization	Data analysis, teaching, resear-ch
PyCharm	Powerful IDE by JetBrains with advanced features	Professional developers, large projects
Spyder	Scientific Python Development Environment, ideal for data science	Data scientists and researchers
Visual Studio Code (VS Code)	Lightweight, extensible editor with Python extensions	Beginners and professionals
IDLE	Default simple IDE that comes with Python installation	Beginners and learning pur-poses

1.1.7.1 Jupyter Notebook

Jupyter Notebook is an open-source, web-based interactive computing environment that allows users to create and share documents containing live Python code, equations, visualizations, and narrative text. It is widely used in data science, machine learning, scientific research, and education because it combines code execution with rich text elements, making it easy to explain and visualize data workflows. Jupyter Notebook work in a browser interface where code is written in cells and executed independently. Users can intersperse code cells with Markdown cells for formatted text, headings, lists, and images. This flexibility makes Jupyter Notebooks ideal for exploratory programming, data analysis, and creating reproducible research documents.

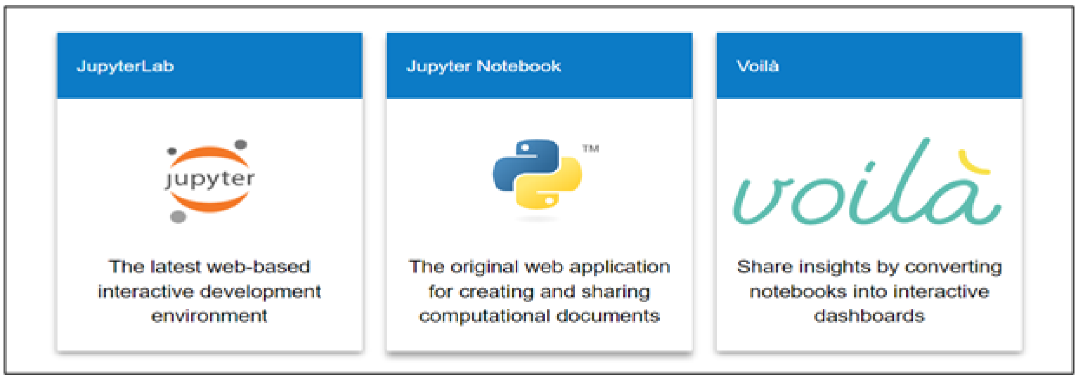
Key advantages of Jupyter Notebook include immediate feedback from code execution, easy visualization of results with libraries like Matplotlib and Seaborn, and the ability to share notebooks with others through various formats such as HTML and PDF. Due to these features, Jupyter Notebook have become a popular tool for both teaching and professional data science work.

Let’s start the journey of learning Python programming by printing a message “hello world “ program. To write a program using the Python programming language, we need an IDE. An integrated development environment (IDE) is an application that provides facilities for software development. IDE consists of an editor to type the program, a facility to highlight the mistakes identified by the IDE, and other features to develop an application without spending much time. A number of IDEs are available for programming in Python. Since we are new to Python programming, let’s start with Jupyter Notebook Online, a simple environment for Python programming. Jupyter Notebooks allows:

- creation and execution of Python programs by integrating code and its output into a single document.
- opening the IDE in a standard web browser.

How to Start?

Step 1: Let’s start Jupyter Notebook Online as shown in fig 1.1.10 by opening the link https://jupyter.org/try.



JupyterLab

The latest web-based interactive development environment

Jupyter Notebook

The original web application for creating and sharing computational documents

Voilà

Share insights by converting notebooks into interactive dashboards

Fig 1.1.10 Jupyter Notebook Online

Step 2: Start a new workbook as shown in fig 1.1.11. We can write the program in the cell provided by the IDE.

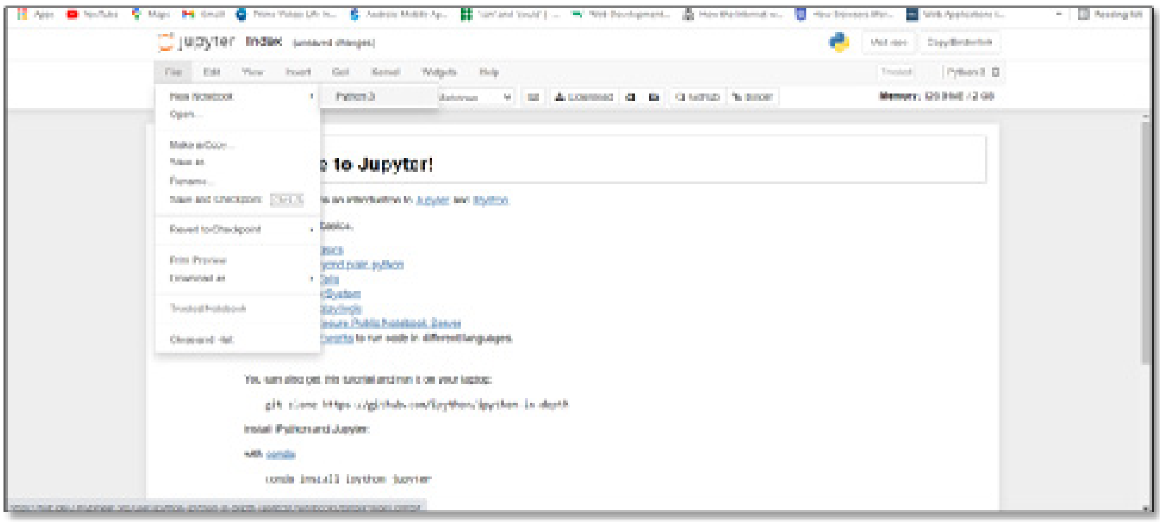


Fig 1.1.11 Start a new workbook

Fig 1.1.12 shows python IDE, here we can write programs.

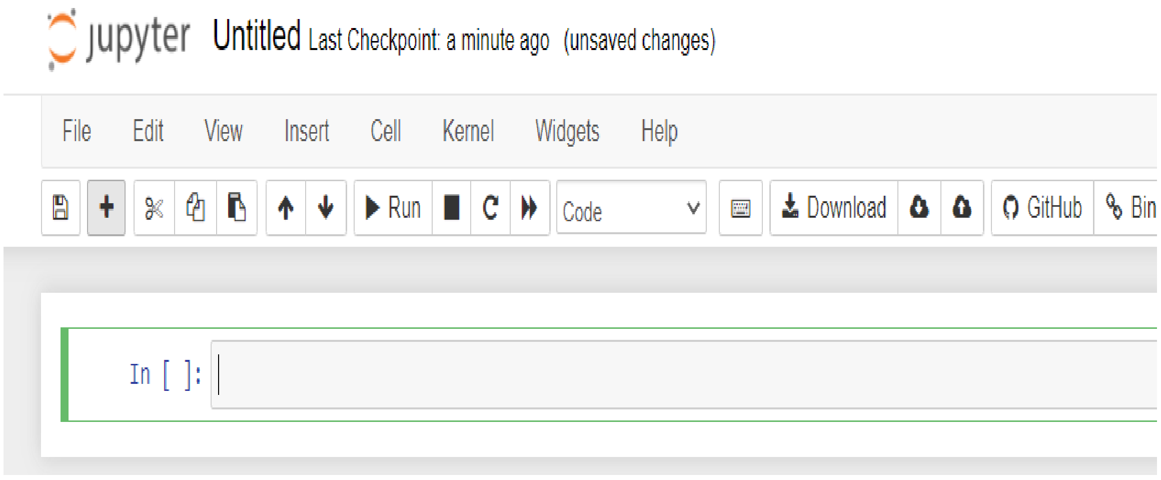
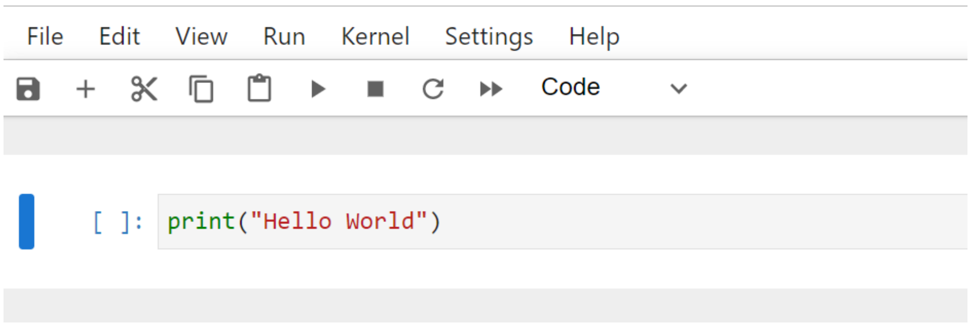

jupyter Untitled Last Checkpoint: a minute ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Run Code Download GitHub Bin
In [ ]:

Fig 1.1.12 Python IDE

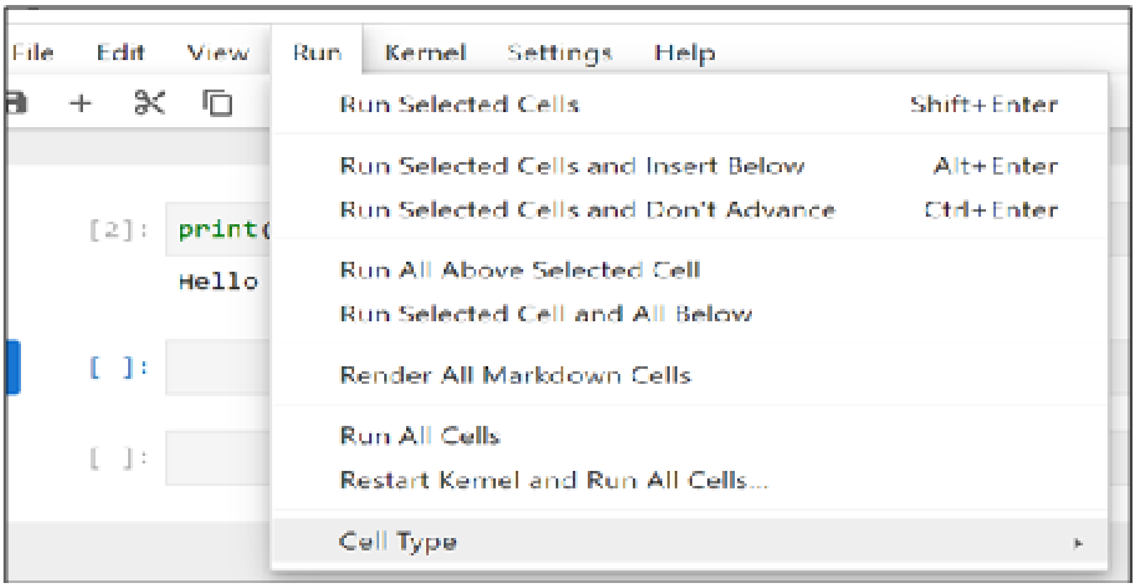
Type print (“Hello World”) as shown in fig 1.1.13 below.



```
[ ]: print("Hello World")
```

Fig 1.1.13 Type the code

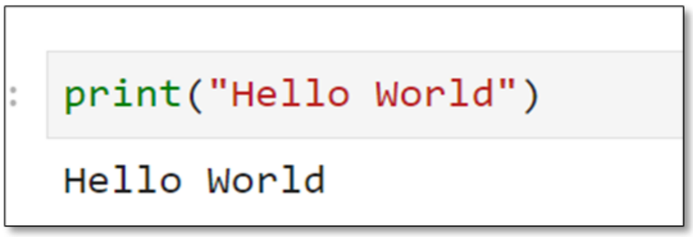
Step 3: Click on the Run button and click on Run Selected Cells or click on ► as shown in fig 1.1.14 to execute the program and observe the result.


File Edit View Run Kernel Settings Help
Run Selected Cells Shift+Enter
Run Selected Cells and Insert Below Alt+Enter
Run Selected Cells and Don't Advance Ctrl+Enter
Run All Above Selected Cell
Run Selected Cell and All Below
Render All Markdown Cells
Run All Cells
Restart Kernel and Run All Cells...
Cell Type

```
[2]: print(
     Hello
[ ]:
[ ]:
```

Fig 1.1.14 Run option in Python IDE

The following fig 1.1.15 shows the result will be displayed.



```
print("Hello World")

Hello World
```

Fig 1.1.15 Result

1.1.7.2 PyCharm

PyCharm is one of the most powerful and widely used Integrated Development Environments (IDE) for Python development. It is developed by **JetBrains** and provides a complete set of tools for professional developers to build, test, debug, and manage Python applications efficiently. It helps developers write clean, efficient, and error-free code by providing intelligent code completion, debugging tools, project navigation, and support for frameworks like Django, Flask, and more. Features of PyCharm is shown in table 1.1.2.

Table 1.1.2: Features of PyCharm

Features	Description
Smart Code Editor	Provides auto-completion, syntax highlighting, and code suggestions.
Debugger and Testing Tools	Integrated visual debugger and test runner.
Version Control Integration	Supports Git, SVN, Mercurial for version control.
Project Navigation	Easily jump to classes, files, functions, or usages.
Refactoring Tools	Rename, move, delete, extract methods, and more without breaking code.
Database Support (Pro)	Built-in tools to manage and query databases.
Framework Support (Pro)	Includes Django, Flask, Pyramid, and other web frameworks.
Cross-platform	Available for Windows, macOS, and Linux.

Creating Your First Project in PyCharm

Step 1: Let's start PyCharm by opening the link https://www.jetbrains.com/pycharm/download

Step 2: When you first use PyCharm, you'll be welcomed by its startup screen (figure 1.1.16).

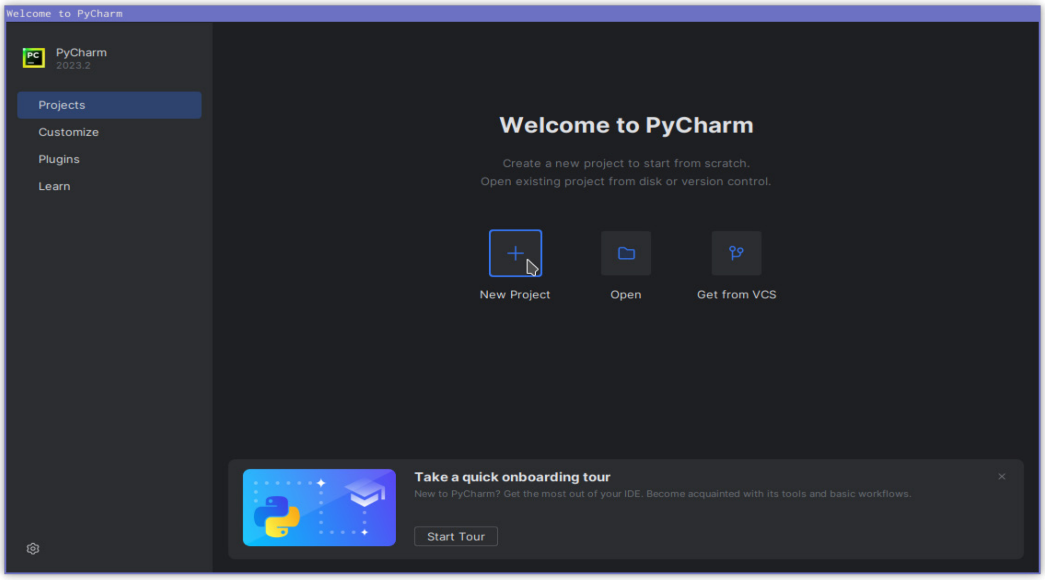
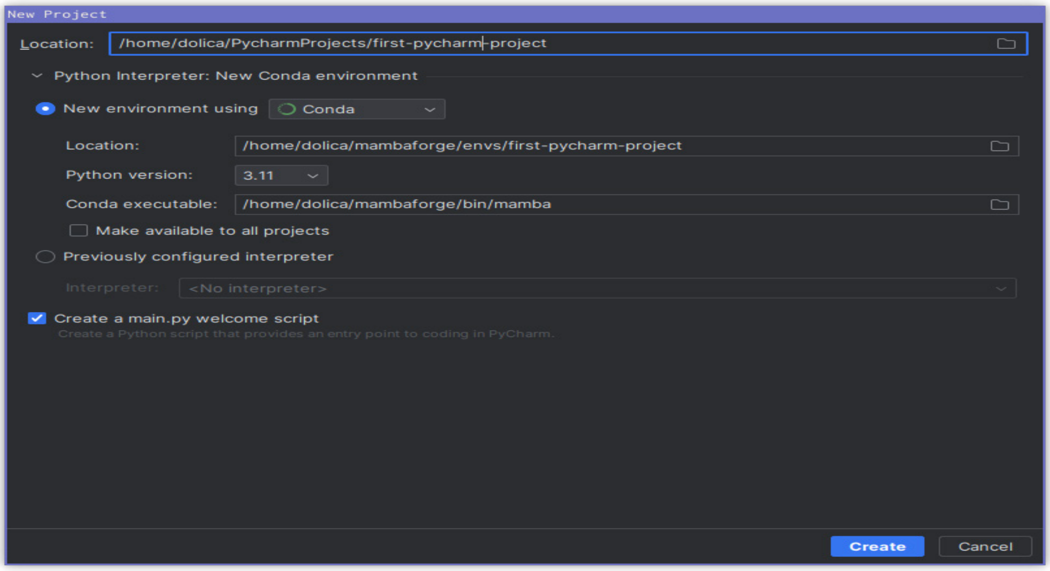

Welcome to PyCharm
PyCharm
2023.2
Projects
Customize
Plugins
Learn
Welcome to PyCharm
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
Take a quick onboarding tour
New to PyCharm? Get the most out of your IDE. Become acquainted with its tools and basic workflows.
Start Tour

Fig 1.1.16 PyCharm startup screen

Step 3: To create a blank Python project, click on "New project." PyCharm will now prompt you to configure the Python project (figure 1.1.17).



New Project

Location: /home/dolica/PycharmProjects/first-pycharm-project

Python Interpreter: New Conda environment

New environment using Conda

Location: /home/dolica/mambaforge/envs/first-pycharm-project

Python version: 3.11

Conda executable: /home/dolica/mambaforge/bin/mamba

Make available to all projects

Previously configured interpreter

Interpreter: <No interpreter>

Create a main.py welcome script

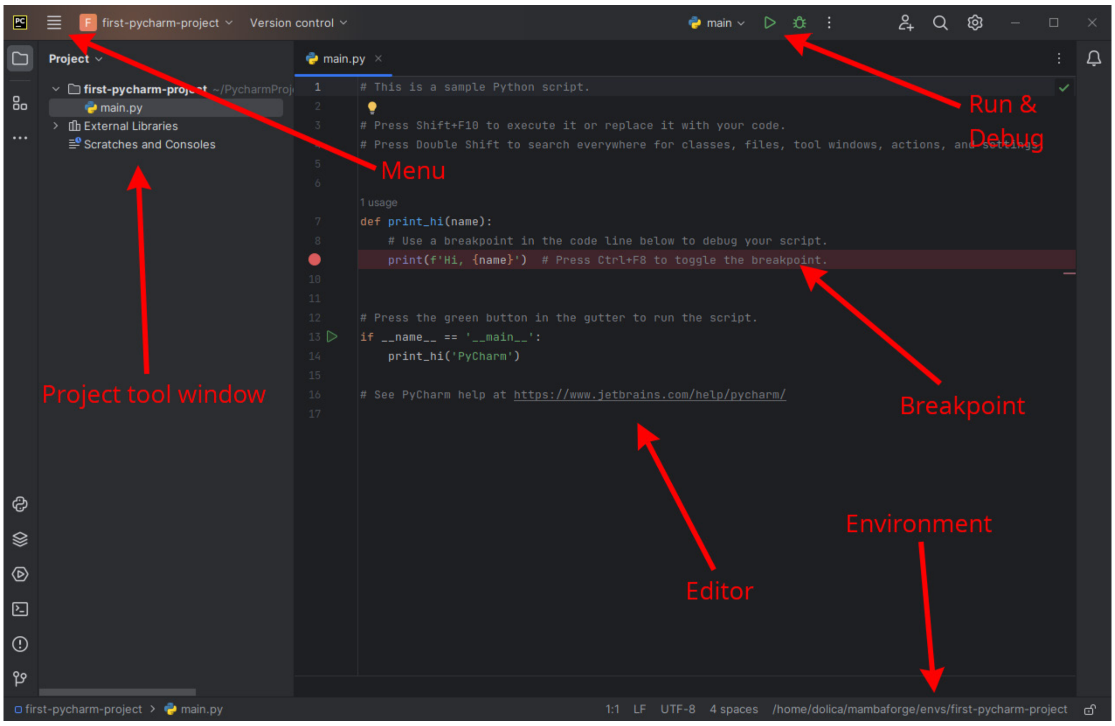
Create a Python script that provides an entry point to coding in PyCharm.

Fig 1.1.17 PyCharm configure the Python project

In this example, the project files will reside in my ***/home/dolica/PyCharmProjects/first-pycharm-project*** directory. However, you can choose a location that suits your needs. For the virtual environment, select Conda. PyCharm automatically makes the virtual environment's name match the project name, and this is something we wish to keep as it makes things less confusing.

The setup page also asks if you want to create a ***main.py*** welcome script; for this project, you should keep this option checked. This will have PyCharm create a basic Python file for you instead of just preparing a blank directory. Once the setup is complete, click ***"Create."***

Step 4: After creating your project, you'll land in the main PyCharm interface (figure 1.1.18).



```python
# This is a sample Python script.

# Press Shift+F10 to execute it or replace it with your code.
# Press Double Shift to search everywhere for classes, files, tool windows, actions, and settings.


1 usage
def print_hi(name):
    # Use a breakpoint in the code line below to debug your script.
    print(f'Hi, {name}')  # Press Ctrl+F8 to toggle the breakpoint.


# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    print_hi('PyCharm')

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

Fig 1.1.18 The main PyCharm interface

Here's a quick overview:

- **Project Tool Window:** This displays the files comprising your Python project. Right now, there's only one file, main.py, but more complex projects will have multiple files and folders.
- **Menu:** Clicking here opens the PyCharm menu.
- **Editor:** This is where you write code. The tab bar at the top lets you switch between different files, though currently, we have only one, main.py.
- **Environment:** This shows the environment used to run your Python files for this project, matching the Mamba settings you selected when creating the project.
- **Breakpoint:** Breakpoints are handy for pausing program execution, especially when debugging code to find issues.
- **Run & Debug:** The run button (play symbol) runs your code and displays the output in the console. The debug button (bug symbol) runs the program in debugging mode, pausing at breakpoints.

Step 5: Now you should see the PyCharm editor with our ***`main.py`*** file open. You'll notice a ***`print_hi`*** command created by PyCharm. You'll learn more about how these commands work and how to create your own later. For now, let's run this file (figure 1.1.19).

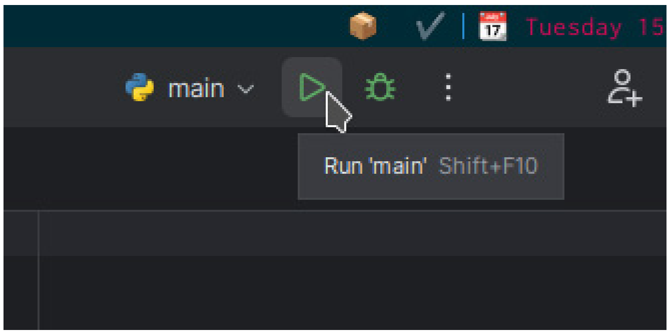
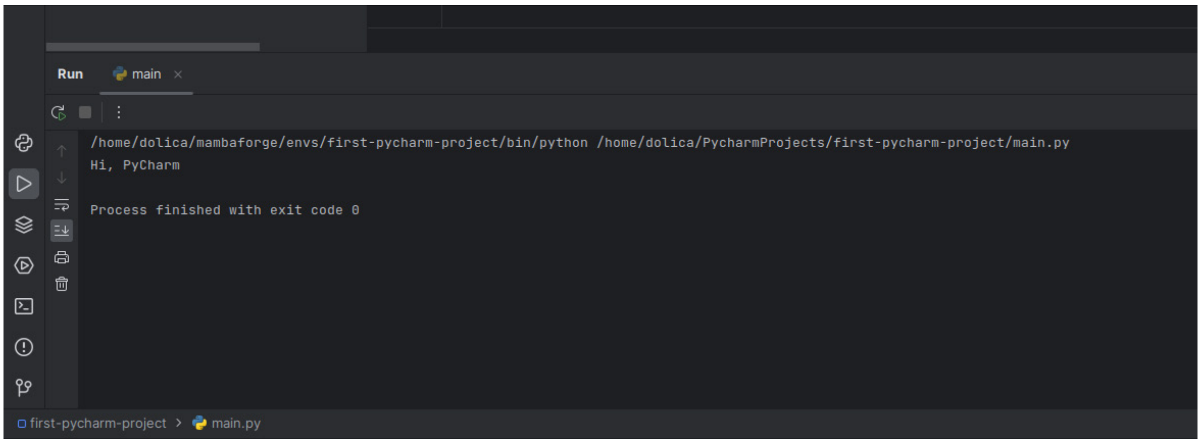


Fig 1.1.19 Run option in PyCharm interface

Step 6: This opens the "Run" panel at the bottom of the window, displaying the output(figure 1.1.20).



```
/home/dolica/mambaforge/envs/first-pycharm-project/bin/python /home/dolica/PycharmProjects/first-pycharm-project/main.py
Hi, PyCharm

Process finished with exit code 0
```

Fig 1.1.20 Displaying the output

The text "Process finished with exit code 0" indicates that the program ran without errors. Our ***`main.py`*** code is designed to display ***"Hi, PyCharm"*** and it has executed correctly.

PyCharm makes Python development faster, easier, and more manageable with its intelligent tools and automation features. Whether you're a beginner or professional, it's an excellent choice for working on Python projects efficiently.

1.1.7.3 Spyder

Spyder is a specialized Python IDE tailored for scientists, engineers, and data analysts. It provides a clean and organized environment for writing, running, and debugging Python code, especially in fields like data science and numerical computing. Spyder stands for **Scientific Python Development Environment**, an open-source platform that integrates essential scientific libraries and tools into a single user interface. It is designed to support efficient analysis, visualization, and computation tasks using Python. Spyder interface is shown below (fig 1.1.21).

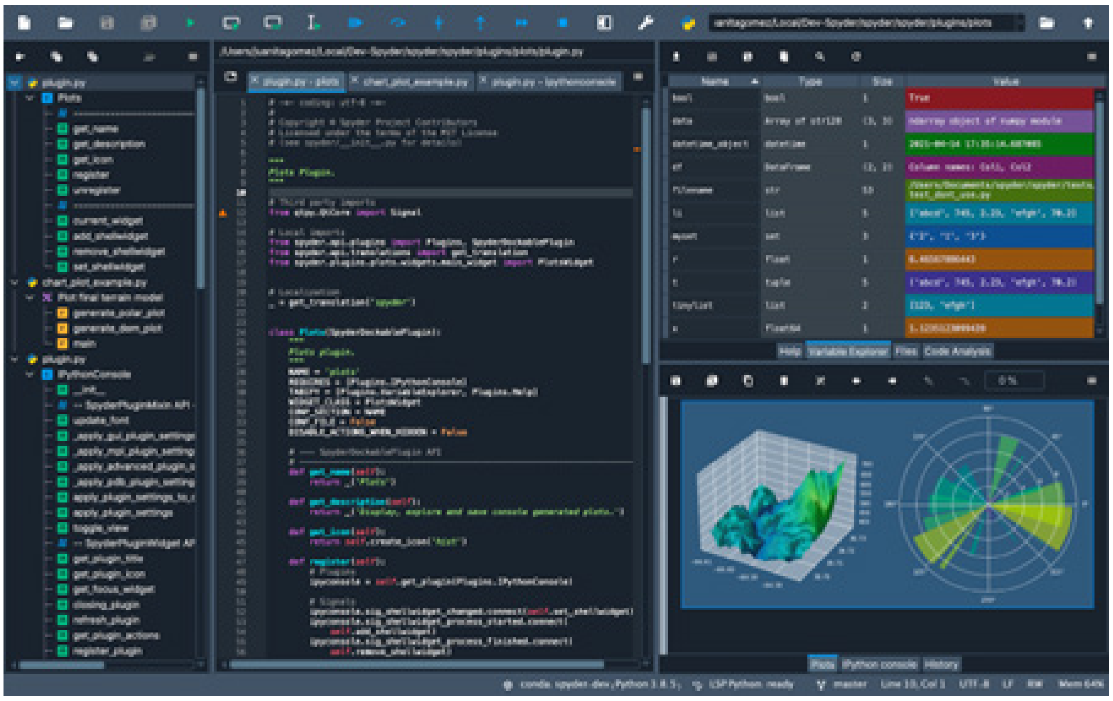


Fig 1.1.21 Spyder interface

Key Features of Spyder

Spyder offers a rich set of tools specifically designed to support scientific computing and data analysis using Python. Its intuitive interface combines code editing, interactive execution, variable tracking, and visualization in a single environment. These features make Spyder especially useful for researchers, data scientists, and students who need a streamlined workflow for numerical computations and exploratory programming. Some of them are:

- **Advanced Code Editor**: Offers syntax highlighting, indentation, and auto-completion for faster coding.
- **IPython Console Integration**: Executes Python commands interactively with immediate output.
- **Variable Explorer**: Displays variables, arrays, and data frames in a spreadsheet-like format.
- **Built-in Profiler**: Measures code performance to identify and optimize slow sections.
- **Static Code Analysis**: Uses tools like Pylint to detect errors and improve code quality.
- **Documentation Viewer**: Shows docstrings and function references in real-time as you type.
- **Multiple Panes Interface**: Organizes editor, console, explorer, and help views for multitasking.

- **Seamless Library Support**: Compatible with scientific libraries like NumPy, Pandas, Matplotlib, and SciPy.
- **Customizable Layout**: Allows users to rearrange and configure panels according to workflow needs.
- **Plugin Support**: Extends capabilities by integrating additional tools and features.

1.1.7.4 Visual Studio Code (VS Code)

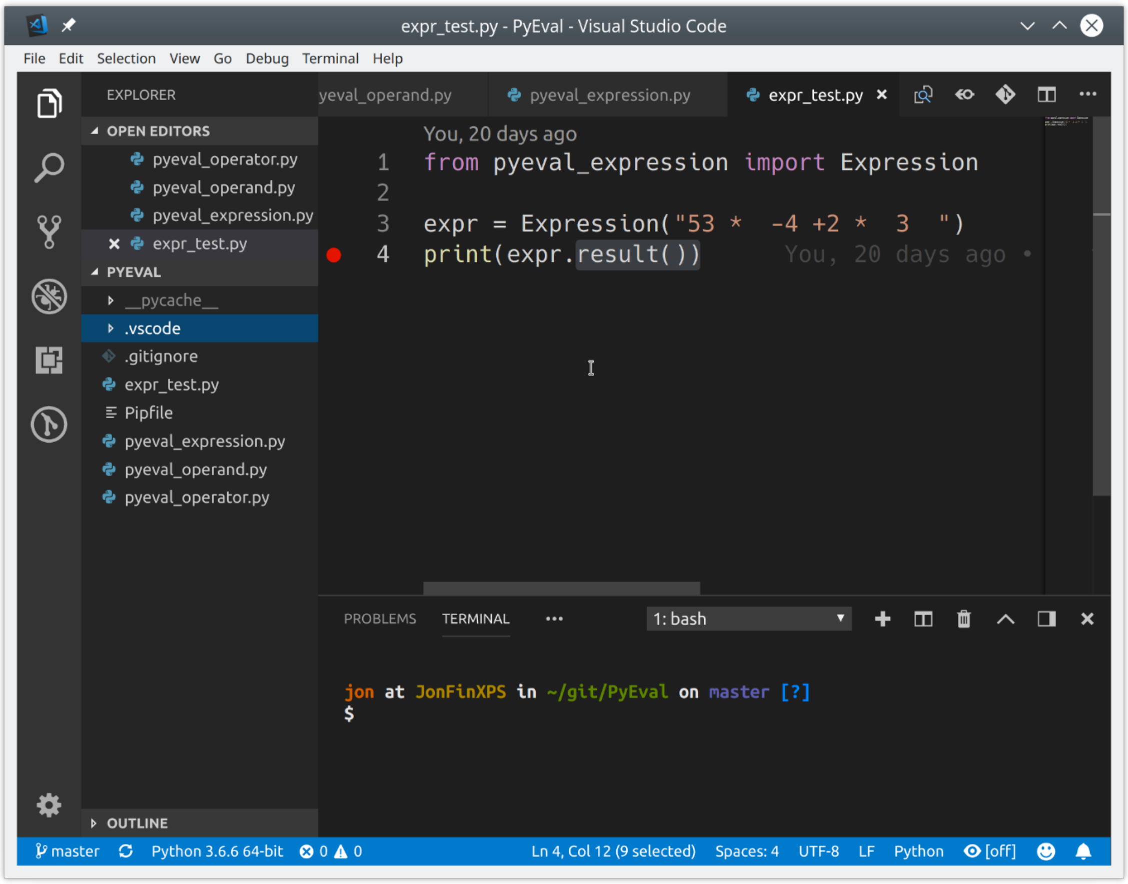
Visual Studio Code is a lightweight and highly customizable code editor developed by **Microsoft**, popular among developers for its speed, flexibility, and broad language support. It is widely used for general-purpose programming, web development, and scripting. **VS Code** is a **free, open-source source code editor** that supports development in various languages like Python, JavaScript, Java, C++, and more. It combines a powerful editing experience with integrated tools for debugging, version control, and extensions (fig 1.1.22).

Key Features of VS Code

Visual Studio Code is a versatile and lightweight code editor that provides a powerful development experience across a wide range of programming languages and platforms. Its extensive feature set includes smart code editing, built-in debugging, and seamless integration with version control systems. With a vast extension marketplace and customizable interface, VS Code adapts easily to the needs of web developers, software engineers, and data scientists alike.

- **Multi-language Support**: Provides syntax highlighting, linting, and IntelliSense for numerous programming languages.
- **Integrated Terminal**: Allows command-line access directly within the editor.
- **Extension Marketplace**: Hosts thousands of plugins to enhance functionality and support additional tools or languages.
- **Built-in Git Integration**: Enables commit, push, pull, and branch operations without leaving the editor.
- **Smart Code Completion**: Suggests relevant code snippets, methods, and variables using IntelliSense.
- **Real-time Debugging**: Offers breakpoints, watch expressions, and call stacks for detailed analysis.
- **Customizable Themes**: Lets users change color schemes, icons, and layout preferences.
- **Remote Development Support**: Enables editing and running code on remote servers or inside containers.
- **Live Share Collaboration**: Allows real-time code sharing and team collaboration directly within the editor.

- ♦ **Minimal System Requirements**: Runs efficiently even on systems with limited resources.



```
from pyeval_expression import Expression

expr = Expression("53 *  -4 +2 *  3  ")
print(expr.result())        You, 20 days ago •
```

```
jon at JonFinXPS in ~/git/PyEval on master [?]
$
```

Fig 1.1.22 VS code interface

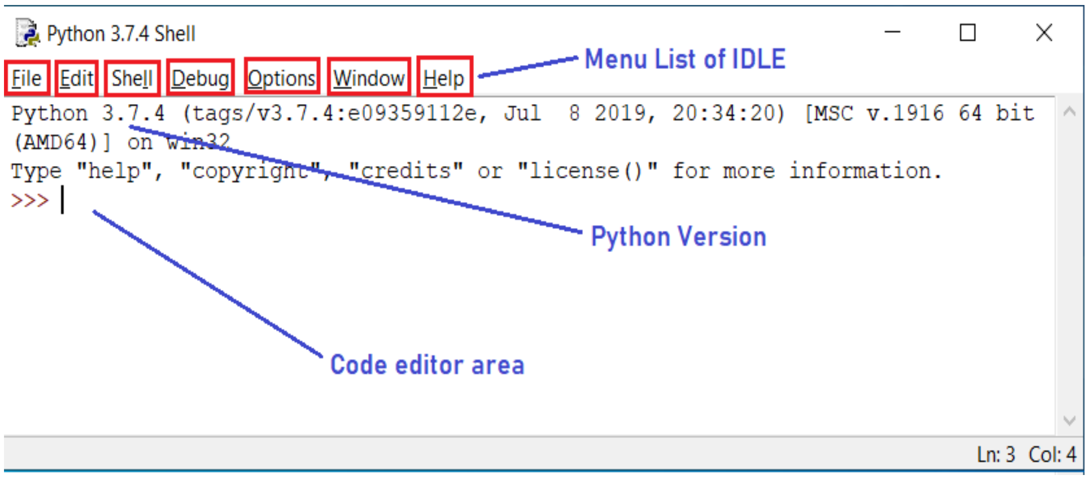
1.1.7.5 IDLE

IDLE is the default development environment that comes bundled with Python, intended for beginners and those who want a simple interface to write and test code. It is designed to be easy to use and requires no additional installation when Python is installed. **IDLE** stands for **Integrated Development and Learning Environment**, a lightweight Python IDE built using the Tkinter GUI toolkit. It provides a basic interface for editing, running, and debugging Python scripts, making it suitable for educational and learning purposes (fig 1.1.23).

Key Features of IDLE

- ♦ **Python Shell**: Offers an interactive interpreter where users can type and execute code line by line.
- ♦ **Simple Code Editor**: Includes syntax highlighting, indentation, and basic code formatting.
- ♦ **Built-in Debugger**: Provides step-through debugging with breakpoints and variable inspection.
- ♦ **Cross-Platform Compatibility**: Runs on Windows, macOS, and Linux with a consistent interface.

- **Auto-Completion**: Suggests functions and variable names while typing.
- **Integrated Help System**: Allows access to Python documentation directly from the interface.
- **Lightweight Installation**: Requires minimal system resources and no configuration.
- **Menu-Driven GUI**: Features standard menus for file, edit, run, and debug operations.
- **Script Execution Support**: Enables running entire Python programs directly from the editor.
- **Educational Focus**: Designed with simplicity in mind, making it ideal for classroom use and self-learning.



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
```

Fig 1.1.23 IDLE interface

This unit provided a foundational understanding of Python and how to set up a development environment for effective programming. It began with the installation process of Python on various operating systems, ensuring learners can run Python programs locally. The unit then introduced Python's interactive mode, a valuable feature for quickly testing code and learning through immediate feedback. Finally, it offered an overview of popular Python IDEs such as Jupyter Notebook, IDLE, PyCharm, Spyder, and VS Code, highlighting their key features and helping learners choose the most suitable tools for their needs. Together, these topics equip beginners with the essential skills to start writing and experimenting with Python code confidently.

Recap

- Python is a high-level, interpreted programming language known for its simplicity and readability.
- Installing Python is the first step to start writing and executing Python code on any computer system.
- Python's **interactive mode** allows users to run individual lines of code and view immediate results.
- Interactive mode is useful for quick testing, debugging, and learning basic Python syntax.
- You can access Python's interactive shell by simply typing `python` or `python3` in the terminal or command line.
- The interactive prompt is indicated by the >>> symbol, where code can be typed and executed directly.
- An **Integrated Development Environment (IDE)** is a software tool that helps write, run, and manage code more efficiently.
- Jupyter Notebook is a web-based, open-source environment for interactive programming and data analysis.
- **IDLE** is Python's built-in IDE, offering a simple interface with a code editor and interactive shell.
- **PyCharm** is a feature-rich IDE with tools for debugging, project management, and web development (Professional edition).
- **Spyder** is ideal for scientific computing and data analysis, offering a variable explorer and integrated IPython console.
- **Visual Studio Code (VS Code)** is a lightweight, cross-platform editor with powerful extensions for Python development.
- IDEs improve productivity by offering features like syntax highlighting, auto-completion, and debugging support.
- Choosing the right IDE depends on the user's needs, such as beginner-friendly use (IDLE), data science (Spyder), or general coding (VS Code, PyCharm).

Objective Type Questions

1. What type of language is Python categorized as?
2. Who developed the Python programming language?
3. In which year was Python first released?
4. What is the primary file extension for a Python script?
5. What command is used to open the Python interactive shell in the terminal?
6. What symbol represents the Python prompt in interactive mode?
7. Which built-in tool comes installed with Python for simple editing and execution?
8. What does IDE stand for?
9. Which IDE is specifically designed for scientific computing and data analysis?
10. What command launches Jupyter Notebook from the command line?
11. Which Python IDE is developed by JetBrains?
12. What is the key debugging tool integrated into most IDEs?
13. Which Python IDE shows variables and data in a spreadsheet-like view?
14. What is the primary use of Markdown in Jupyter Notebooks?
15. Which IDE is known for being highly customizable with extensions?
16. What is the default GUI toolkit used by IDLE?
17. Which IDE is considered best for collaborative real-time coding using Live Share?
18. What Python environment is recommended for beginners due to its simplicity?
19. Which command-line option adds Python to system-wide use on Windows during installation?
20. Which IDE is accessible entirely through a web browser interface?
21. What does the “Run” button in most IDEs do?
22. What is the purpose of the Python interactive mode?
23. Which tool allows users to mix code, plots, and text in a single document?
24. Which key feature makes VS Code a cross-platform editor?
25. Which command-line tool is used to install Python packages?

Answers to Objective Type Questions

1. High-level interpreted language
2. Guido van Rossum
3. 1991
4. `.py`
5. `python` or `python3`
6. `>>>`
7. IDLE
8. Integrated Development Environment
9. Spyder
10. `jupyter notebook`
11. PyCharm
12. Debugger
13. Spyder
14. To format text and add explanations
15. Visual Studio Code (VS Code)
16. Tkinter
17. Visual Studio Code (Live Share extension)
18. IDLE
19. Add Python to PATH
20. Jupyter Notebook
21. Executes the selected code or script
22. To test and run Python commands one line at a time
23. Jupyter Notebook
24. It works on Windows, macOS, and Linux with the same core features
25. pip

Assignments

1. Explain why Python is considered a high-level programming language. Give two examples of features that support this classification.

2. Describe the step-by-step process to install Python on a Windows and Linux operating systems. Include the significance of adding Python to the system PATH for Windows operating system.

3. What is Python interactive mode? Demonstrate with an example how you can use this mode to perform simple arithmetic operations.

4. Compare and contrast the following Python IDEs (Jupyter Notebooks, PyCharm, Spyder, and VS Code). Highlight their key features and mention which types of users each IDE is best suited for.

Reference

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

2. Sweigart, A. (2015). *Automate the boring stuff with Python* (1st ed.). No Starch Press.

3. VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. O'Reilly Media.

4. Grinberg, M. (2018). *Flask web development: Developing web applications with Python*. O'Reilly Media.

5. Pilgrim, M. (2009). *Dive into Python 3*. Apress.

Suggested Reading

1. Python Software Foundation. (n.d.). *Python documentation*. https://docs.python.org/3/

2. Project Jupyter. (n.d.). *Jupyter Notebook documentation*. https://jupyter-notebook.readthedocs.io/en/stable/

3. JetBrains. (n.d.). *PyCharm: The Python IDE for Professional Developers*. https://www.jetbrains.com/pycharm/

4. Spyder IDE. (n.d.). *Spyder Documentation*. https://docs.spyder-ide.org/

5. Microsoft. (n.d.). *Visual Studio Code documentation*. https://code.visualstudio.com/docs/languages/python

Unit 2
Variables, Data Types, and Basic Input/ Output

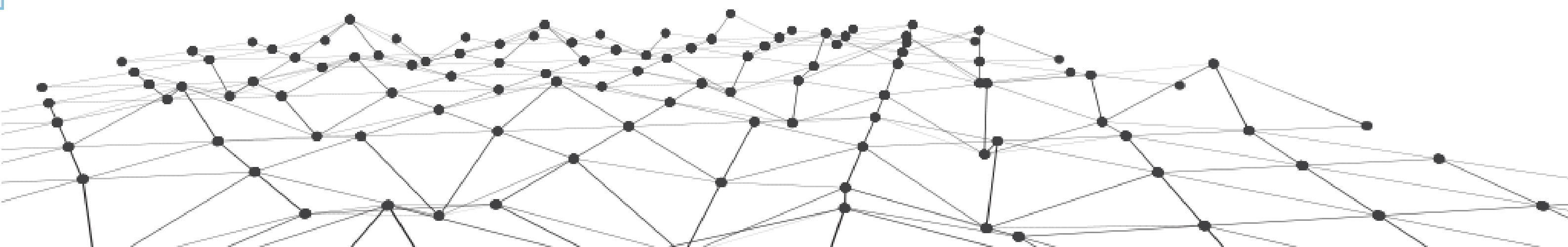
Learning Outcomes

After completing this unit, the learner will able to:

- understand how to declare and use variables in Python.
- differentiate between basic Python data types like int, float, str, and bool.
- use input() and print() functions for user interaction.
- perform type casting to convert between different data types
- follow proper variable naming rules in Python.

Prerequisites

Imagine you're designing a software system for a school to manage student records, attendance, and exam results. You need a way to store a student's name, age, subjects, and scores. This is where variables and data types come into play. Variables act as labeled storage boxes to hold information, and data types tell the computer what kind of data is being stored, like whether it's text (name), numbers (marks), or a combination of both. Without clearly understanding variables and the right data types, a program would be unable to correctly process or store data, leading to errors or incorrect outputs. Additionally, programs often need to interact with users, like taking inputs for a student's details or displaying final grades. This interaction is made possible through input and output operations in Python. For example, a user might enter a score using the keyboard, and the program might respond by printing the student's grade. Mastering variables, data types, and input/output operations is essential not just for building programs that work, but also for making them user-friendly, dynamic, and responsive. These concepts form the foundation for everything in Python, from simple calculators to complex data analysis and AI systems.



Key words

variable, data type, input, output, typecasting, boolean, dictionary

Discussion

Programming is all about working with data, storing it, processing it, and interacting with users. In Python, the most basic yet essential tools for handling data are **variables**, **data types**, and **input/output operations**. These three concepts form the foundation of every Python program, whether it's a simple calculator or a complex AI model.

A **variable** in Python is like a container or a label that holds a value. It allows us to store data in memory so we can use it later in the program. Python is a **dynamically typed** language, which means you don't need to declare the type of a variable when you create it.

Python figures it out automatically based on the value you assign. This makes the language easier to learn and use, especially for beginners.

Data types tell the computer what kind of value is being stored in a variable. Python supports several standard data types such as **integers** for whole numbers, **floats** for decimal numbers, **strings** for text, and **booleans** for true/false logic. Choosing the right data type is important because it determines what operations you can perform on the data.

Python also allows interaction between the program and the user through **input and output**. Using the input() function, we can take data from the user during program execution. The print() function, on the other hand, allows us to display messages or results to the screen. Together, these functions make our programs interactive and user-friendly.

variables, **data types**, and **input/output** operations are the core tools that enable a Python program to handle information. Mastering these concepts is essential for writing correct, efficient, and meaningful code.

1.2.1 Variable

Imagine you have a bunch of empty boxes, and you want to put different items inside them. You'd probably label each box so you know what's in it, right?

In Python, these "boxes" are called variables. They are named spots in the computer's memory where you can store different pieces of information. How to Use a Variable (Putting Stuff in a Box): You give your box a name, then use the = sign to put a value inside it.

1.2.1.1 Variable Declaration in Python

In Python, **declaring a variable** simply means assigning a value to a name using the equals sign =. Python is a **dynamically typed** language, so you don't need to mention the type of the variable it is automatically understood from the value assigned. Example,

x = 10

This line creates a variable named x and stores the number 10 in it. Python automatically detects the type based on the value (this is called **dynamic typing**).

If you want to specify the data type of a variable, this can be done with casting.

x = str(3)   # x will be '3'

y = int(3)   # y will be 3

z = float(3) # z will be 3.0

Variables can store different types of data, such as:

- Numbers (like 10, 3.14)
- Text (like "Hello")
- Boolean values (True, False)

You can change the value of a variable anytime in your program. You can also assign values to multiple variables at once:

a, b, c = 1, 2, 3

You can even assign the same value to multiple variables:

x = y = z = 100

1.2.1.2 Rules for Naming Variables in Python

Python has specific rules and best practices for naming variables. These rules help avoid errors and make the code readable and understandable:

1. Start with a letter or underscore (_)

A variable name must begin with a letter (A–Z, a–z) or an underscore _.

Valid: total, _value Invalid: 1score

2. Followed by letters, digits, or underscores

After the first character, you can use any combination of letters, digits, or underscores.

Valid: marks_1, user_name

3. No special characters or spaces

Variable names cannot contain symbols like @, $, #, or spaces.

Invalid: user-name, total marks

4. Cannot be a Python keyword

Keywords like if, while, for, class, and def cannot be used as variable names.

5. Case-sensitive

Python treats uppercase and lowercase letters as different.

For example, Age, age, and AGE are all different variables.

6. Use meaningful names

It is a good habit to use clear and descriptive names that explain the purpose of the variable.

Good: student_name, item_price

Bad: x, abc (unless used in short loops)

1.2.2 Data types in Python

Data is processed by applications or programs. For example, in the student registration process, we are using different data such as name, date of birth, address, family monthly income, etc. Different types of data are used in the registration process. Date of birth is date type, monthly income is numeric data, a name is a group of letters.

A data type is a classification of data. Data type is a type of data or variable which is used in programs. Memory location will be allocated according to the type of data. For example, the memory requirement of storing the values "KKG",5, and 5.10 will be different. Python supports different data types as detailed below:

1.2.2.1 Numeric data types

Numeric data types in Python represent the data containing numeric values. Numbers can be described in various ways, including floating, integers, or complex numbers.

Python has three numeric data types: float, integer, and complex. Float stores decimal numbers, integers can store whole numbers, and complex is for storing complex numbers.

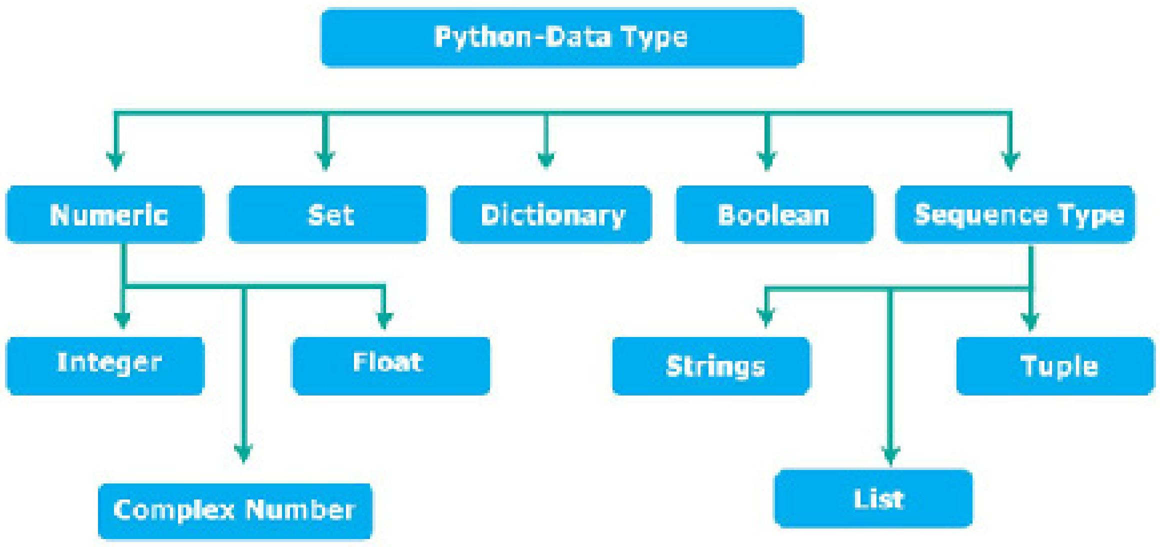


Fig 1.2.1 Data types

1. Integers (int)

Integers represent both positive and negative whole numbers without any fractional or decimal components. One of the advantages of integers in Python is that they can be of unlimited length. Additionally, Python allows you to improve the readability of large integers by using underscores in place of commas. For example, the number one million can be written as 1000000 or as 1_000_000.

Integers can be Octal and Hexadecimal also. 0o32 represents an octal number. 0x32 represents Hexadecimal. Note that the first digit is zero followed by the letter o for octal and x for hexadecimal

Example:

```
a= 90

b=hex(a)

print(b)

#Output: 0x5a
```

Run the above program using bin() and oct() functions.

bin represents the binary number and oct represents the octal number

2. Float

The float class in Python is used to represent real numbers that include a decimal point, commonly referred to as floating-point numbers. These numbers can be either positive or negative and are typically accurate up to 15 decimal places. The presence of a decimal point distinguishes a float from an integer.

Python also allows float values to be written in **scientific notation**, where the letter E or e is used to indicate that the number should be multiplied by 10 raised to a certain power. This format is known as **exponential notation**. For example, the number one million can be represented as 1000000.0, 1_000_000.0, or 1e6.

Example : 4.5,890.67

3. Complex numbers

The complex class represents a complex number data type. Complex numbers are expressed as (real part) + (imaginary part) j, where j denotes the imaginary unit. These numeric data sets are primarily used in computer graphics and scientific computing. Complex numbers are used in geometry, scientific calculations, and calculus.

Example: 3+4j

1.2.2.2 Sequence Data Type

In Python, **sequence data types** refer to ordered collections of elements, which can be of the same or different data types. These data structures are designed to store multiple values efficiently and allow users to access, modify, or process elements using indexing and slicing techniques.

Python’s sequence types are broadly categorized into three groups:

- **Basic sequence types:** list, tuple, and range
- **Text sequence type:** str (string)
- **Binary sequence types:** bytes, bytearray, and memoryview

Each of these types serves a specific purpose and supports various operations such as iteration, indexing, slicing, and membership tests. While some sequences like strings and tuples are **immutable** (cannot be changed after creation), others like lists and bytearrays are **mutable** (can be changed).

In the following sections, we will focus on three commonly used sequence data types in Python **string**, **list**, and **tuple** and explore their features along with examples.

1. String data type

A string in Python is a sequence of characters. Characters are letters, numbers, symbols, etc. Strings are used when you need to process text data like names, addresses, etc.

Example: “Hello World”

“Covid-19”

Data in between the quotes “ ” are string data.

“222345” is not a number, it is a string.

Different types of string representation

a. # single quotes string

Message = ‘Hello World’

b. # double quotes string

Message = “Hello World”

c. # triple quotes for multiline strings. Three single quotes or double quotes can be used.

Message = ‘” Programming is fun. Python is a high-level language. Python is used by Facebook, Google, and other companies ‘”

When we input the data from the keyboard, the number will be considered as string only.

2. List

In Python, a **list** is an ordered and mutable sequence used to group related data items together. Lists can store elements of **any data type**, including numbers, strings, or even other lists. One of the key features of lists is their **mutability**, which means that you can modify, add, or remove items after the list has been created without changing its identity. Lists are typically created using **square brackets** [ ], with items separated by commas. They support a wide range of operations, including indexing, slicing, appending, removing, and iterating through elements. Since lists maintain the order of insertion, each element is accessible by its index, starting from 0 for the first item.

x = [“apple”, “grapes”, “cherry”]

In this example, x is a list containing three string elements.

3. Tuple

A **tuple** in Python is also an ordered sequence, much like a list, but with one important difference: tuples **are immutable**. This means that once a tuple is created, its elements cannot be altered, added to, or removed. Due to this property, tuples are often used for **fixed collections of items** where data integrity is important. Tuples can store elements of various types and support indexing and iteration just like lists. They are typically defined by placing values separated by **commas**, and are often enclosed in **parentheses** (), though the parentheses are optional in many cases.

x = (“apple”, “grapes”, “cherry”)

Here, x is a tuple containing three elements. Unlike lists, any attempt to modify the tuple after creation will result in an error.

1.2.2.3 Boolean Data Types

The **Boolean** data type in Python is a built-in type that represents one of two possible truth values: **True** or **False**. These values are particularly important in control flow operations such as conditionals (if, while) and logical expressions. In Python, any object can be evaluated in a Boolean context. Values such as non-zero numbers, non-empty strings, and collections are interpreted as **True**, while zero, None, empty sequences (“”, [], {}), and other “empty” types are considered **False**. You can explicitly create a Boolean value using the bool() constructor, which converts a given value to either True or False based on its truthiness. Booleans are actually a subclass of integers in Python, with True having the value 1 and False having the value 0.

Example:

X= True

Result = False

1.2.2.4 Set Data Type

A **set** in Python is a built-in data structure that represents an **unordered collection of unique elements**. Unlike lists and tuples, sets do not maintain any specific order, and they automatically eliminate duplicate entries. Sets are **mutable**, meaning their contents can be changed after creation by adding or removing elements. However, Python also provides an **immutable version** of sets called frozenset, which cannot be altered once created. A set can contain mixed data types, such as strings, integers, or even tuples, as long as the elements themselves are **hashable**. Sets can be created either by using the built-in set() function with an iterable or by enclosing comma-separated elements within curly braces {}. They support various set operations such as **union (|)**, **intersection (&)**, and **difference (-)**, which make them particularly useful in tasks involving membership testing, duplicate removal, and mathematical computations.

Example: x = {“apple”, “grapes”, “cherry”}

1.2.2.5 Dictionary Data Type

A **dictionary** in Python is an **unordered collection of key-value pairs**, where each key is unique and maps to a specific value. Dictionaries are also known as **associative arrays** or **hash maps** in other programming languages. They are used to store data values like a real-world dictionary, where words (keys) are associated with their meanings (values). A key in a dictionary must be immutable (such as a string, number, or tuple), while values can be of any data type and can be duplicated. Dictionaries are created using **curly braces {}** with key-value pairs separated by **colons :**, and each pair is separated by a **comma**. Dictionary keys are **case-sensitive**, meaning “Name” and “name” would be treated as different keys. To access a value, you refer to its key either directly using square brackets [] or safely using the get() method, which prevents errors if the key doesn’t exist. Dictionaries are versatile and widely used for representing structured data such as JSON objects, configuration files, and datasets.

Example:

x = {“name”: “Rose”, “age”: 16}

1.2.3 Input & Output

This is how your program shows you things and how you can give information to your program. The date given to the computer to process is called Input and when Computer Process the input data and provide the result, that result is called Output.

When we need to provide input to python programs, Python provides some statements called Input statements and when it wants to display some output, it can also be done by Output statements.

1. Output ()

For Output statements Python uses **print ()** functions, which can be used in a variety of ways to display programs output.

Example 1: Displaying a Simple Message

print(“Hello, Python learner!”) # Output: Hello, Python learner!

We can use the print() function to print single and multiple variables. We can print multiple variables by separating them with commas.

Example:

# Single variable

s = “Bob”

```
print(s)
```

# Multiple Variables

```
s = “Alice”

age = 25

city = “New York”

print(s, age, city)
```

#Output

```
Bob

Alice 25 New York
```

2. Input ()

Python uses **input ()** functions to take input from the keyboard. This function takes value from the keyboard and returns as a string.

```
name = input(“Enter your name: “)

print(“Hello,”, name, “! Welcome!”)
```

#Output:

```
Enter your name: Priya

Hello, Priya ! Welcome!
```

We are taking multiple inputs from the user in a single line, splitting the values entered by the user into separate variables for each value using the split() method. Then, it prints the values with corresponding labels, either two or three, based on the number of inputs provided by the user.

# taking two inputs at a time

```
x, y = input(“Enter two values: “).split()

print(“Number of boys: “, x)

print(“Number of girls: “, y)
```

# taking three inputs at a time

```
x, y, z = input(“Enter three values: “).split()

print(“Total number of students: “, x)

print(“Number of boys is : “, y)

print(“Number of girls is : “, z)
```

#Output

```
Enter two values: 4 11
Number of boys:  4
Number of girls:  11
Enter three values: 8 7 12
Total number of students:  8
Number of boys is :  7
Number of girls is :  12
```

Activity 1: Run the following program and check the output

```
mark = input("Enter a mark")
print(mark)
print(type(mark))
```

#Input:

```
Enter a  mark: 67.9
```

# Output:

```
67.9
<class 'str'>
```

To read a number as a float number, Python uses Typecasting to convert one data type to another.

Example

```
mark = float(input("mark"))
print(mark)
print(type(mark))
```

#Input

```
67.9
```

#Output

```
67.9
<class 'float'>
```

The second method of data type casting is mark = float(mark)

Example

```
a =int(input("enter first number"))
b =int(input("enter second number"))
c =float( a) + float(b)
print(c)
#Input
enter first number: 10
enter second number: 20
#Output
30.0
```

Recap

- Variables are named storage for values used in a program.
- Python uses dynamic typing, meaning variable types are inferred automatically.
- Common data types include int, float, str, bool, list, tuple, set, and dict.
- The input() function collects user data as a string.
- The print() function displays output to the screen.
- Typecasting changes one data type into another (e.g., str to float).
- Lists and tuples store sequences; lists are mutable, tuples are not.
- Sets hold unique, unordered elements; dictionaries store key-value pairs.
- Boolean data types hold values of either True or False.
- split() is used to take multiple inputs at once from a single line.

Objective Type Questions

1. What keyword is used to take input from the user in Python?
2. Which data type is used to store a whole number?
3. What function is used to display output in Python?
4. What data type is returned by the input() function?
5. Which data type can store a sequence of characters?
6. What keyword is used to define a variable in Python?
7. Which data type has only two possible values: True or False?
8. Which data structure in Python is used to store key-value pairs?
9. What is the result type of int() + float() in Python?
10. Which built-in function is used to convert string to float?

Answers to Objective Type Questions

1. input
2. int
3. print
4. str
5. str
6. No keyword *(Python uses dynamic typing)*
7. bool
8. dict
9. float
10. float

Assignments

1. Define a variable in Python. Write a program that takes your name, age, and city as input and prints them.

2. Explain the difference between a list and a tuple with examples.

3. Write a Python program that accepts three float numbers from the user and calculates their average.

4. What is typecasting? Give an example where typecasting is necessary in a program.

5. Create a dictionary with three keys: name, grade, and percentage. Print the values using the dictionary keys.

6. Take two inputs in one line and print whether the first number is greater than the second using Boolean logic.

Reference

1. Lutz, M. (2021). *Learning Python* (5th ed.). O'Reilly Media.

2. Python Software Foundation. (2024). *The Python Tutorial*. https://docs.python.org/3/tutorial/

3. Beazley, D. M., & Jones, B. K. (2023). *Python Cookbook* (3rd ed.). O'Reilly Media.

4. Van Rossum, G., & Drake, F. L. (2024). *The Python Language Reference Manual* (latest ed.). Python Software Foundation.

5. Oliphant, T. E. (2023). *A Guide to NumPy* (2nd ed.). CreateSpace Independent Publishing Platform.

Suggested Reading

1. Sweigart, A. (2023). *Automate the Boring Stuff with Python* (2nd ed.). No Starch Press.

2. Shaw, Z. A. (2023). *Learn Python the Hard Way* (4th ed.). Addison-Wesley.

3. Slatkin, B. (2023). *Effective Python: 90 Specific Ways to Write Better Python* (2nd ed.). Addison-Wesley.

4. Matthes, E. (2023). *Python Crash Course* (3rd ed.). No Starch Press.

5. McKinney, W. (2023). *Python for Data Analysis* (3rd ed.). O'Reilly Media.

Unit 3

Operators and Expressions

Learning Outcomes

After completing this unit, the learner will able to:

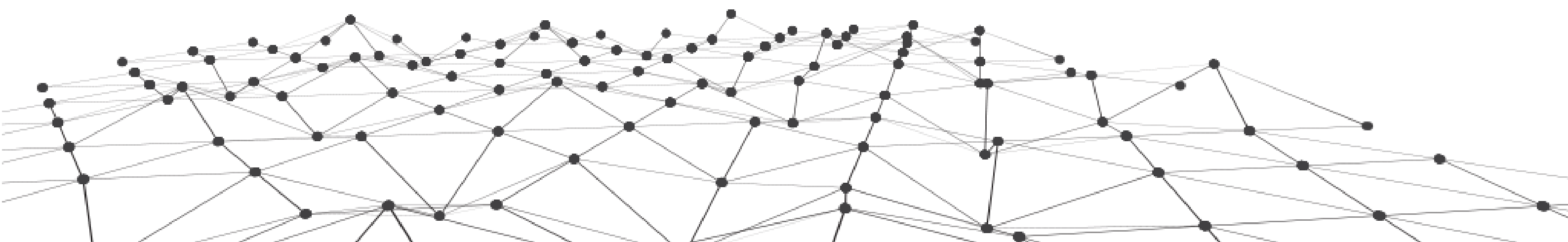
- familiarize different types of operators used in Python programming.
- demonstrate the use of arithmetic, assignment, and comparison operators in expressions and statements.
- analyze logical and bitwise operations to control the flow and manipulate binary data.
- distinguish between identity, membership, and equality operators through practical coding examples.
- apply operator precedence rules to evaluate complex expressions accurately.

Prerequisites

Before learning operators and expressions in Python, it is essential to have a basic understanding of variables, data types, and how to write simple Python programs. Variables store data, and operators allow us to perform actions on that data. For example, in a banking application, if a user withdraws money from an account, the program needs to subtract the withdrawal amount from the account balance. Here, subtraction is performed using the minus operator. Without operators, we cannot perform any meaningful calculations or logical decisions in programs. Expressions combine variables and operators to produce a result, like balance = balance - amount. Operators are also crucial in decision-making, such as checking if a user has enough balance in his account using a comparison operator. In real-life scenarios, in order to log in a system, logical operators help verify both username and password. Thus, understanding operators and expressions is fundamental to building functional and interactive Python applications.

Key words

Operand, Assignment, Arithmetic, Logical, Comparison, membership, Identity



Discussion

1.2.1 Python Operators

When we withdraw money from an ATM, the amount withdrawn will be deducted from the account. The program will subtract the amount. Subtracting amount is an operation and the operator used is minus (-).

Operators are special symbols in a programming language that carries out arithmetic, logic, and other operations. The value or data that the operator operates on is called the operand.

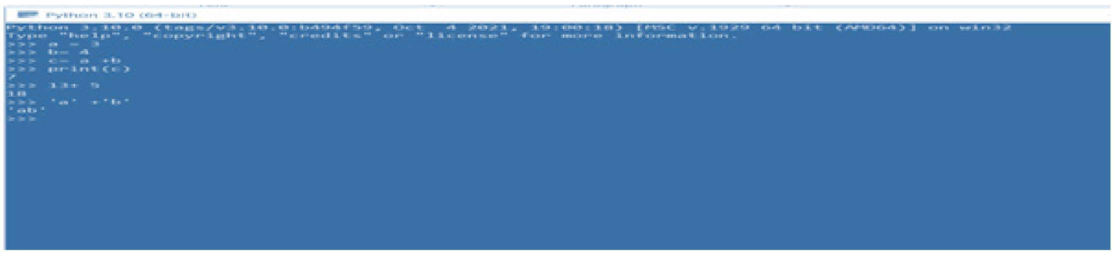
The following are the few types of operators in Python.

- Arithmetic Operators
- Assignment Operators
- Comparison (Relational) Operators
- Logical Operators
- Increment/Decrement Operator
- Bitwise operators
- Membership and Identity

1.2.1.1 Arithmetic Operators

While making applications or programs, sometimes, we need to do some calculations such as addition, subtraction, etc. For example, in an ATM application, the amount will be subtracted when you withdraw money. Arithmetic operators are used to perform such operations.

Note: Use Python interactive mode to practice all examples in this unit. We can use IDE also. In Interactive mode, we can type the instructions and get them done one by one as shown in the figure 1.2.1



```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 3
>>> b= 4
>>> c= a +b
>>> print(c)
7
>>> 13+ 5
18
>>> 'a' +'b'
'ab'
>>>
```

Fig 1.2.1 Python Interactive IDE

Activity 1: Type 13+ 5 in the shell and press enter to see the result.

The following are the arithmetic operators.

- **Operator: + (plus)**

Purpose: + operator is used to add two objects

Example : 13 + 5 = 18 .

```
A=4
B= 4
C= A + B
```

Here C= A + B is an arithmetic expression or statement that uses + as an arithmetic operator and A, B are operands.

- **Operator: - (minus)**

Purpose: To subtract one number from the other; if the primary operand is absent it's assumed to be zero. For example, use -5 instead of writing 0 – 5

Activity : loan = 500

```
          paid = 200
 balance = loan – paid
print(balance)
```

- **Operator: * (multiply)**

Purpose: To multiply two numbers or to replicate a string

Example of multiplication : 20 * 3= 60

Example of string réplication :

'KKG' * 3 = 'KKGKKGKKG', KKG is a string.

- **Operator: / (divide)**

Purpose: Divide one number by another number.

12/ 3= 4

11/3 = 3.6666666666666665

- **Operator: ** (power)**

Purpose: Returns x to the power of y

Example: 3 **2 = 9

- **Operator: // (divide and floor)**

Purpose: Divide m by n and round the answer *down* to the nearest **integer** value.

Note that if one of the values is a float, the result also will be a float.(floats are decimal numbers)

Example:

11 // 3 = 3

-11 // 3 = -4

11.81//1.2 = 9

- **Operator: % (modulo)**

Purpose: Returns the remainder after division

10 % 3 gives 1, 21.9%3 =0.8999999999999986

1.2.1.2 Assignment Operator

- **Operator: = (assign)**:

Purpose: To assign and store a value on the right side of the statement to left side operand

Example: a=50 will assign 50 to the variable name a. a is an operand and 50 is value assigned.

mark =mark +10

Student_Name = "KKG"

Multiple assignments:

Purpose: To assign different values to more than one variable.

It is possible to assign the same value to multiple variables.

Example: m=n=z=10 (assigns value 10 to m, n and z)

Example: x, y, z = 100, "hello", 30.5 (assigns 100 to x, "hello" to y, and 30.5 to z)

1.2.1.3 Shortcut Operators

- **Operator: += (Add and assign)**

Purpose: To perform the addition operation first then do the assignment

Example: mark+=10, equivalent to mark = mark+ 10, suppose the mark is 80, 80 will be added to 10 and the new mark will be 90.

- **Operator: -=, *=, /=, //=, %=, **=**

Example: mark*=2, equivalent to mark = mark*2

salary/=10 is equivalent to salary = salary/10

1.2.1.4 Comparison Operators

While writing programs or applications we will use comparison operators. For example, the ATM application will check the entered amount is less than or equal to(<=) the available amount. If the result is true, you will get money, otherwise, the machine will inform you that you do not have a sufficient amount in the account. All comparison operators will compare the data and return True or False.

- **Operator: < (less than)**

Purpose: To check whether the value of the left side operand is less than the value of the right-side operand. x < y

Example: 3 < 50 gives True .

50 < 3 gives False

3 < 50 < 70 gives True.

3<0<70 gives False

- **Operator: > (greater than)**

Purpose: To check whether x is greater than y

Example: 50 > 3 returns True , 3>50 returns False

- **Operator: <= (less than or equal to)**

Purpose: To check whether x is less than or equal to y

```
x = 3
y = 6
x <= y returns True
```

- **Operator: >= (greater than or equal to)**

Purpose: To check whether x is greater than or equal to y

```
x = 4
y = 3
x >= 3 returns True
```

- **Operator:** == (equal to)

Note: Two equal symbols together without space

Purpose: Compares if the objects are equal

```
x = 20
y = 20
x == y returns True
```

x = ‘KKG’;   y = KKG;   x == y returns True

x = ‘KKG’;   y = ‘kkg’;   x == y returns False. Note that Python is case sensitive.

- **Operator: != (not equal to)**

  m = 20;   n = = 3;   m != 2 returns True

1.2.1.5 Logical Operators

Two or more relations that compare the data can be logically joined together using the logical operators ***or*** and ***and***. For example, an application will check if the username is correct and the password is correct when you log in.

- Operator**: and**

Return True or False based on the conditions.

Table 1.2.1

X	**Y**	**X and Y**
True	True	True
True	False	False
False	True	False
False	False	False

Example :

m = False;

n = True;

m **and** n return False

Operator**: or**

Return True or False based on the conditions.

Table 1.2.2

X	**Y**	**X or Y**
True	True	True
True	False	True
False	True	True
False	False	False

If x is True , it returns True, otherwise it returns evaluation of y

x = True; y = False;   x **or** y returns True.

Note: x & y should be Boolean, otherwise it will return the integer

- Operator: not (boolean NOT)

Purpose: The negation of a Boolean is the opposite of its current Boolean value.

Example

Not(True) means False. Note: The first letter of **T**rue and **F**alse must be a capital letter

```
x= True
y = not(x)
print(y) returns False
```

1.2.1.6 Increment/Decrement Operators

If you have prior experience with Python, you might be aware that it doesn't support traditional increment (++) and decrement (--) operators, whether in pre or post form. This design choice is intentional, aimed at promoting simplicity and code readability. In languages that support these operators, beginners often confuse the differences between pre-increment/decrement and post-increment/decrement, especially regarding operator precedence and return values. However, in Python, such operators are not essential. In this section, we will explore how to implement increment and decrement operations effectively in Python.

Python Increment Operator (+=)

Python uses the += compound assignment operator to increment a value. This operator adds the right-hand operand to the left-hand variable and assigns the result back to that variable. It provides a concise way to update a variable's value.

Unlike other languages where you might see:

```
for (int i = 0; i < 5; ++i)
```

In Python, instead of using i++, which is invalid syntax, you can simply write:

```
i += 1  # or i = i + 1
```

Example:

```
# Initialize a variable
x = 5
# Increment x by 1
x += 1  # Equivalent to x = x + 1
# Print the result
print("Incremented value:", x)
```

Output:

```
Incremented value: 6
```

Python Decrement Operator (-=)

Python also doesn't support the -- decrement operator. However, you can decrease a variable's value using the -= operator. This operator subtracts the right-hand operand from the left-hand variable and stores the result back in that variable.

Instead of using i--, which is invalid in Python, you can write:

```
i -= 1  # or i = i - 1
```

Example:

```
# Initialize a variable

x = 10

# Decrement x by 1

x -= 1  # Equivalent to x = x - 1

# Print the result

print("Decremented value:", x)
```

Output:

```
Decremented value: 9
```

1.2.1.7 Bitwise Operators_

Python provides bitwise operators to manipulate integer data at the binary level. These operators first convert the given integers into their binary form, apply the operation bit-by-bit or on corresponding pairs of bits, and then return the result as a decimal value.

Note: Bitwise operators in Python operate exclusively on integer data types.

Operator	Description	Syntax
&	Bitwise AND	x & y
`|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x >> y
<<	Bitwise left shift	x << y

Bitwise AND Operator

The Bitwise AND (&) operator in Python takes two integers, compares their binary forms, and performs AND on each pair of bits. The result bit is 1 only if both bits in the pair are 1; otherwise, it is 0.

Example: Let X = 7 = $(111)_2$ and Y = 4 = $(100)_2$. Then, X & Y = $(100)_2$ = 4.

```
a = 10

b = 4

print("a & b =", a & b)
```

Output:

```
a & b = 0
```

Bitwise OR Operator

The Bitwise OR (|) operator compares each bit of two integers and returns 1 in each position where at least one of the bits is 1. If both bits are 0, the resulting bit is 0.

Example: X = 7 = $(111)_2$ and Y = 4 = $(100)_2$. Then, X | Y = $(111)_2$ = 7.

```
a = 10

b = 4

print("a | b =", a | b)
```

Output:

```
a | b = 14
```

Bitwise XOR Operator

Bitwise XOR (^), or Exclusive OR, returns 1 if the bits in the compared positions are different, and 0 if they are the same. It is useful when toggling bits.

Example: X = 7 = $(111)_2$ and Y = 4 = $(100)_2$. Then, X ^ Y = $(011)_2$ = 3.

```
a = 10

b = 4

print("a ^ b =", a ^ b)
```

Output:

```
a ^ b = 14
```

Bitwise NOT Operator

The Bitwise NOT (~) operator is unary and acts on a single operand. It inverts each bit — turning 1s into 0s and vice versa. In Python, ~x equals -(x + 1) due to how negative numbers are represented in two's complement.

Example: X = 5 = $(101)_2$. Then, ~X results in the binary inverse, which corresponds to -6.

```
a = 10
```

```
b = 4
print(“~a =”, ~a)
```

Output:

```
 ~a = -11
```

Bitwise Shift Operators

These operators move bits to the left or right. A left shift multiplies the number by powers of two, while a right shift divides the number by powers of two, discarding bits from one end and filling with zeros or sign bits depending on the number’s sign.

Bitwise Right Shift (>>)

Right shifting a number moves its bits to the right. For positive numbers, zeros are inserted from the left; for negative numbers, the sign bit is preserved.

Example:

```
a = 10     # Binary: 0000 1010
b = -10    # Binary: 1111 0110 (in two’s complement)
print(“a >> 1 =”, a >> 1)
print(“b >> 1 =”, b >> 1)
```

Output:

```
a >> 1 = 5
b >> 1 = -5
```

Bitwise Left Shift (<<)

Left shifting a number moves its bits to the left and appends zeros to the right. This effectively multiplies the number by powers of two.

Example:

```
a = 5      # Binary: 0000 0101
b = -10    # Binary: 1111 0110
print(“a << 1 =”, a << 1)
print(“b << 1 =”, b << 1)
```

Output:

```
 a << 1 = 10
 b << 1 = -20
```

1.2.1.8 Membership and Identity Operators

Python offers a wide variety of operators that can be applied to different data types. At times, we may need to check whether a particular value exists within a collection like a list, string, or dictionary. To handle such tasks, Python provides Membership Operators and Identity Operators. This explanation explores both types of operators and how they work.

Membership Operators in Python

Membership operators are used to check whether a value is present in a sequence (like a string, list, tuple, or dictionary). Python provides two membership operators: in and not in.

a. The in Operator

This operator checks if a specified element is found within a sequence. It returns True if the element exists, otherwise it returns False.

```
list1 = [1, 2, 3, 4, 5]

str1 = “Hello World”

dict1 = {1: “Geeks”, 2: “for”, 3: “geeks”}

print(2 in list1)        # True

print(‘O’ in str1)        # False (case-sensitive)

print(3 in dict1)        # True (checks for key, not value)
```

The not in Operator

This operator returns True if the value is not present in the sequence and False otherwise.

```
print(2 not in list1)     # False

print(‘O’ not in str1)    # True

print(3 not in dict1)     # False
```

b. Using operator.contains() Method

As an alternative to in, Python provides the contains() function from the operator module. This function checks if a value exists in a sequence. It takes two arguments — the sequence and the element to search for.

```
import operator

print(operator.contains([1, 2, 3, 4, 5], 2))                # True

print(operator.contains(“Hello World”, ‘O’))               # False

print(operator.contains({1, 2, 3, 4, 5}, 6))                # False

print(operator.contains({1: “Geeks”, 2: “for”, 3: “geeks”}, 3))    # True
print(operator.contains((1, 2, 3, 4, 5), 9))                    # False
```

Identity Operators in Python

Identity operators are used to check whether two variables refer to the exact same object in memory, not just whether their values are equal. Python provides two identity operators: is and is not.

a. The is Operator

This operator returns True if both operands refer to the same object (i.e., they share the same memory location).

```
num1 = 5

num2 = 5

a = [1, 2, 3]

b = [1, 2, 3]

c = a

s1 = "hello world"

s2 = "hello world"

print(num1 is num2)   # True (integers with same value may be stored at same location)

print(a is b)        # False (same values but different objects)

print(a is c)        # True (c is a reference to a)

print(s1 is s2)       # True (string interning)
```

b. The is not Operator

This operator returns True if the variables do not refer to the same object in memory.

```
print(num1 is not num2)   # False

print(a is not b)        # True

print(a is not c)        # False

print(s1 is not s2)       # False

print(s1 is not s1)       # False
```

Difference Between == and is

A common confusion among learners is between the == and is operators. The == operator checks if the values of two variables are equal, while is checks whether the two variables point to the same object in memory.

```
a = [1, 2, 3]

b = [1, 2, 3]

print(a is b)   # False (different objects)

print(a == b)   # True (equal values)
```

In this example, although both lists a and b contain the same elements, a is b returns False because they are two separate objects. However, a == b returns True since the contents are identical.

Order of Operations

In an expression with more than one operator, the order of execution of operators depends on the rules of precedence. Expressions in parentheses are executed first.

Python Operator Precedence and Associativity Table

This table presents Python operators in order from highest to lowest precedence, along with their type, and how expressions involving them are grouped (associativity).

Table 1.2.3

Precedence	Operators	Purpose	Associativity
1	()	Group expressions using parentheses	Left to right
2	x[index], x[index:index]	Indexing and slicing of sequences	Left to right
3	await x	Await expression (used in asynchronous code)	Not applicable
4	**	Exponentiation (power calculation)	Right to left
5	+x, -x, ~x	Unary positive, negative, bitwise NOT	Right to left
6	*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, modulo	Left to right
7	+, -	Addition and subtraction	Left to right
8	<<, >>	Bitwise shift operators	Left to right
9	&	Bitwise AND	Left to right
10	^	Bitwise XOR (exclusive OR)	Left to right
11	`	Bitwise OR	Left to right

12	in, not in, is, is not, <, <=, >, >=, !=, ==	Comparison, membership, identity tests	Left to right
13	not x	Logical NOT	Right to left
14	and	Logical AND	Left to right
15	or	Logical OR	Left to right
16	if ... else	Conditional expressions	Right to left
17	lambda	Lambda (anonymous function) expression	Not applicable
18	:=	Assignment expression (walrus operator)	Right to left

Example 1:

x = 2

y = 4

z = x + y / 2

print(z) will display 4.0. In this expression + and / are the operators. y/2 will be executed first and the result will be added to x.

Example 2:

x = 2

y = 4

z = (x+y)/2

print(z) will display 3.0. In this expression + and / are the operators. Expressions in parentheses are evaluated first. x + y will be executed first and the result will be divided by 2.

Example 3:

x = 2

y = 4

z = x*5 > y

print(z) will return True

Congratulations! You've reached the end of unit 2. Here's the recap of the objectives we have covered and practiced.

Recap

- Operators are special symbols that perform operations on values or variables (operands) in Python.
- Arithmetic operators perform mathematical calculations like addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), floor division (//), and modulo (%).
- The assignment operator (=) assigns the value on the right to the variable on the left.
- Compound assignment operators like +=, -=, *=, /=, //=, %=, **= combine an operation with assignment.
- Comparison (relational) operators such as <, >, <=, >=, ==, != are used to compare values and return True or False.
- Logical operators include and, or, and not, and are used to combine or invert Boolean expressions.
- Python does not support ++ or -- increment/decrement operators; instead, use += 1 or -= 1.
- Bitwise operators operate on binary values using &, |, ~, ^, <<, and >>.
- Bitwise AND (&) returns 1 only if both bits are 1; used for masking bits.
- Bitwise OR (|) returns 1 if at least one bit is 1; used for setting bits.
- Bitwise XOR (^) returns 1 if bits are different; useful for toggling bits.
- Bitwise NOT (~) inverts each bit and returns -(x + 1) in Python.
- Membership operators in and not in check if a value exists within a sequence like list, tuple, string, or dictionary keys.
- Identity operators is and is not check whether two variables refer to the same object in memory.
- == checks if values are equal, whereas is checks if the objects are identical (same memory address).
- Operator precedence defines the order in which operations are performed in an expression; parentheses () have the highest precedence.
- Associativity determines how operators of the same precedence are grouped (left-to-right or right-to-left).
- Example: In x + y / 2, division happens before addition due to higher precedence of /.
- In Python interactive mode or IDE, you can test each operator and expression

step-by-step for better understanding.

- Python promotes readability by avoiding confusing increment/decrement syntax and encourages clear use of compound assignment operators.

Objective Type Questions

1. Which operator is used for exponentiation in Python?
2. What is the result of 11 // 3 in Python?
3. Which operator is used to check object identity in Python?
4. What is the result of 10 % 3?
5. What does the += operator do?
6. Which arithmetic operator is used for division in Python?
7. What will be the output of True and False?
8. Which bitwise operator is represented by ^?
9. Which operator is used to check for membership in a sequence?
10. What is the symbol for assignment operator in Python?
11. What does not True return?
12. Which operator is used to compare if two values are equal?
13. What is the output of 3 < 50 < 70?
14. What is the precedence level of the ** operator?
15. Which logical operator returns True if at least one operand is true?
16. What is the result of a = 5; a += 1? (final value of a)
17. Which operator is used to shift bits to the right?
18. What is the output of ~5 in Python?
19. Which function from the operator module checks for membership?
20. Which operator is used to assign the same value to multiple variables?

Answers to Objective Type Questions

1. **
2. 3
3. is
4. 1
5. AddAssign
6. /
7. False
8. XOR
9. in
10. =
11. False
12. ==
13. True
14. 4
15. or
16. 6
17. >>
18. -6
19. contains
20. =

Assignments

1. Explain the different types of arithmetic operators in Python with suitable examples.
2. Describe the role of assignment and shortcut assignment operators in Python. Provide code examples for each.

3. What are comparison operators in Python? Explain how they are used to compare values and return Boolean results with examples.

4. Explain the difference between identity operators and equality operators in Python with examples.

5. Discuss the purpose and working of logical operators in Python. How does Python evaluate conditions using and, or, and not?

6. Write a note on bitwise operators in Python. Explain each operator with binary examples and show their usage in Python code.

Unit 4

Control Structures

Learning Outcomes

After completing this unit, the learner will able to:

- define control structures used in Python.
- list the types of conditional statements and loops in Python.
- recall the syntax of if, if-else, and elif statements.
- identify the keywords used for for and while loops.
- explain the purpose of break and continue statements in loops.

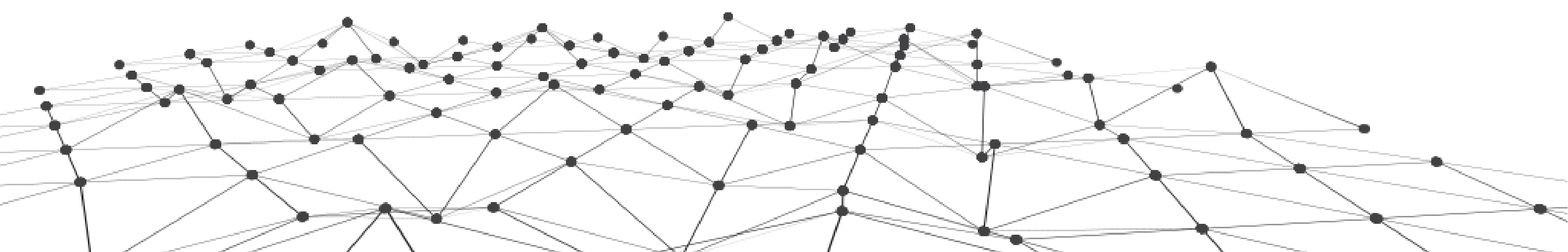
Prerequisites

You already know that a computer program runs instructions one after the other. But what happens when we want the program to make a decision or repeat something many times? For example, think about daily routines like "If it's Monday, go to college" or "Repeat this exercise 10 times." These everyday decisions and repetitions are just like what we can do in programming using control structures.

In this topic, we will learn how to use if-else statements to make decisions and loops like while and for to repeat actions. By linking this to what you already understand about the sequence of instructions in a program, you'll now be able to make your Python code smarter and more flexible. This knowledge will help you solve a wide range of problems more effectively.

Key words

loops, for, while, if-else statements, break and continue



Discussion

1.4.1 Control Structures

Control structures help the program decide what to do and repeat actions when needed just like we do in real life. For example, we might say, "If I'm hungry, I'll eat. In the same way, control structures in Python guide the computer on when to run certain blocks of code and how many times.

There are two main types of control structures in Python:

1. Conditional Statements- used for making decisions
2. Loops - used for repeating actions

1.4.2 Conditional Statements

Conditional statements are used to make decisions in a program based on conditions (True or False). They help the computer choose between different actions depending on the situation. For example, if a student scores more than 50 marks, the program can print "Pass"; otherwise, it can print "Fail". Python uses keywords like if, if-else, and elif to write such conditions.

1.4.2.1 if statement

The if statement in Python is used to execute a block of code only when a specified condition is true. It allows the program to make decisions during execution by checking whether a given condition holds. If the condition evaluates to true, the statements within the if block are executed; otherwise, the program skips that block and continues with the next instructions.

Syntax:

```
if condition:

   # statements to execute if condition is true
```

Flowchart:

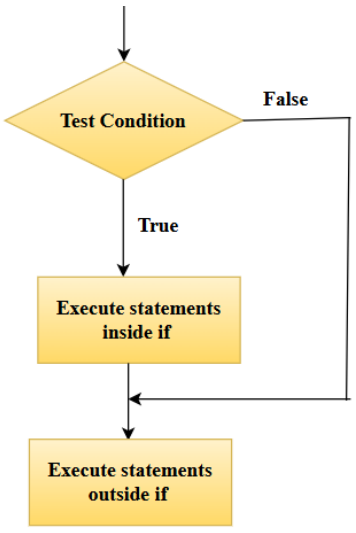


Fig. 1.4.1 flowchart of if statement

Example

```
number = 10
if number > 5:
    print("Number is greater than 5")
```

The output of the above program is:

```
Number is greater than 5
```

1.4.2.2 if else statements

In an if-else statement, the condition provided in the if part is first evaluated. If the condition is True, all the statements under the if block are executed. If the condition is False, the control moves to the else block, and the statements written under else are executed.

Syntax:

```
if condition:
    # statements to execute if condition is True
else:
    # statements to execute if condition is False
```

Flowchart:

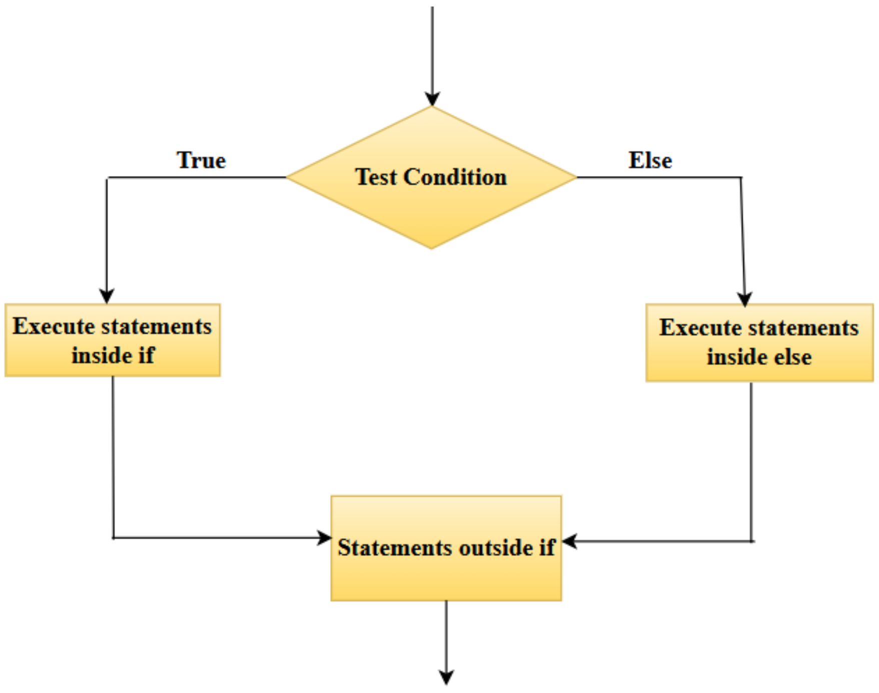


Fig 1.4.2 flowchart of if-else statements

Example:

```
age = 18
if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible to vote")
```

Output:

```
Eligible to vote
```

1.4.2.3 Nested if statements

A nested if-else statement refers to an if or if-else statement placed inside another if or if-else block. It allows the program to make a series of decisions in a hierarchical manner. This structure can be implemented in two common ways:

1. By placing an if statement inside the if block of another if statement.
2. By placing an if statement inside the else block of an if-else statement.

Syntax of method 1:

```
if condition1:
    # statements for condition1 is True
    if condition2:
        # statements for condition2 is True
    else:
        # statements for condition2 is False
else:
    # statements for condition1 is False
```

Syntax of method 2:

```
if condition1:
    # statements for condition1 is True
else:
    if condition2:
        # statements for condition2 is True
    else:
        # statements for condition2 is False
```

Flowchart:

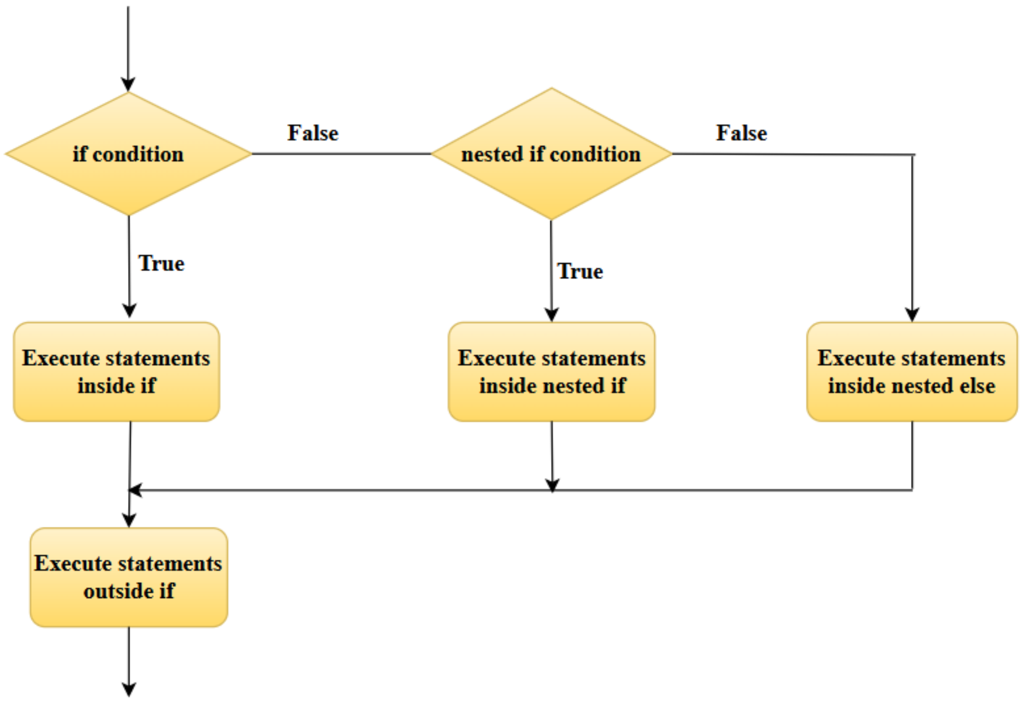

if condition
False
nested if condition
False
True
True
Execute statements inside if
Execute statements inside nested if
Execute statements inside nested else
Execute statements outside if

Fig 1.4.3 flowchart of nested if statement

Example:

```
x = 10
if x >= 0:
    if x == 0:
        print(“Zero”)
    else:
        print(“Positive number”)
else:
    print(“Negative number”)
```

Output:

```
Positive number
```

1.4.3.4 elif ladder statements

The elif ladder, short for “else if”, is a control structure used when multiple conditions need to be evaluated in sequence. It simplifies the logic of complex decision-making, where several possible outcomes exist. Instead of writing multiple nested if-else blocks, the elif ladder offers a clean, readable, and structured way to handle multiple conditional branches.

1. The program first checks the condition in the if statement.
2. If it evaluates to True, the corresponding block is executed, and the rest of the ladder is skipped.
3. If it is False, the program checks the next elif condition, and so on.
4. If none of the if or elif conditions are true, the optional else block is executed.

Syntax:

Syntax:

```
if expression:
        statement(s)
elif expression:
        statement(s)
elif expression:
        statement(s)
else:

        statement(s)
```

Flowchart:

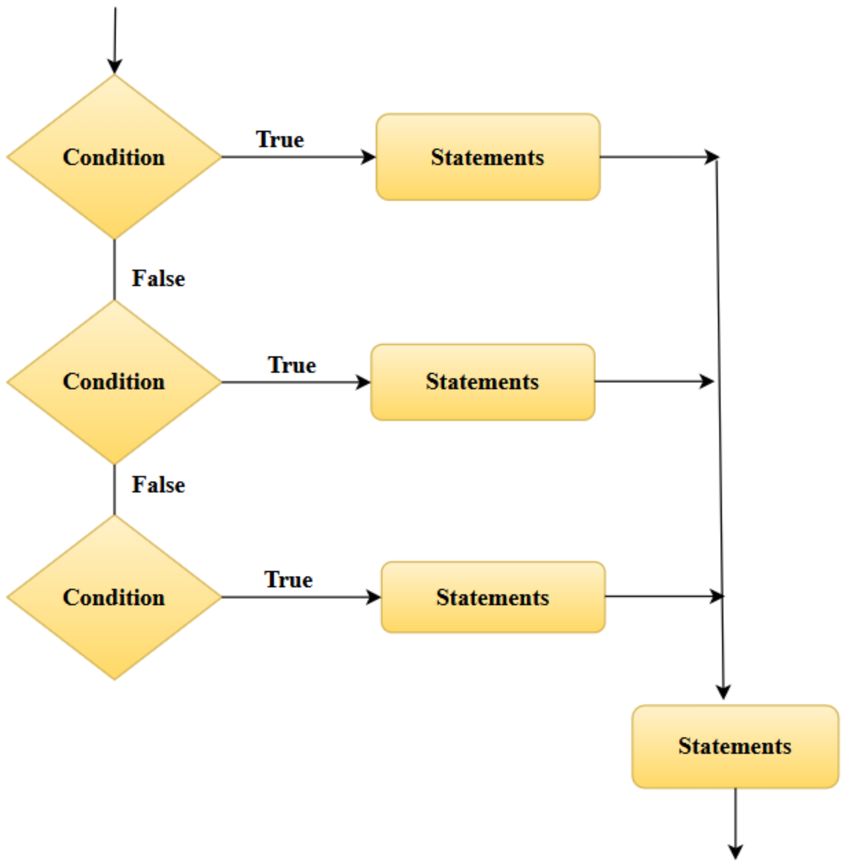


Fig 1.4.4 flowchart of elseif ladder statement

Example:

```
x= -2
if x > 0:
    print("Positive number")
elif x == 0:
    print("Zero")
else:
    print("Negative number")
```

Output:

```
Negative number
```

1.4.3 Loops

A loop is a control structure that allows a block of code to be executed repeatedly as long as a specific condition is satisfied. Loops are used when we want to perform repetitive tasks such as processing items in a list, printing patterns, or running a block of code multiple times.

Python supports two main types of loops:

1. for loop
2. while loop

1.4.3.1 for Loop

A for loop is used when we want to repeat a group of instructions a specific number of times. It is especially useful when we know in advance how many times we want to run the loop. In Python, the for loop is often used with a collection of items, such as a list, a tuple, or a string. These collections are called iterable objects because we can go through each item one by one. At the beginning of the loop, Python picks the first item in the collection and assigns it to a loop variable. Then, it runs the block of code inside the loop. This process continues for each item in the collection until all the items have been used. Once the sequence is finished, the loop stops automatically.

Syntax

```
for variable in sequence:
  statement(s)
```

Flowchart:

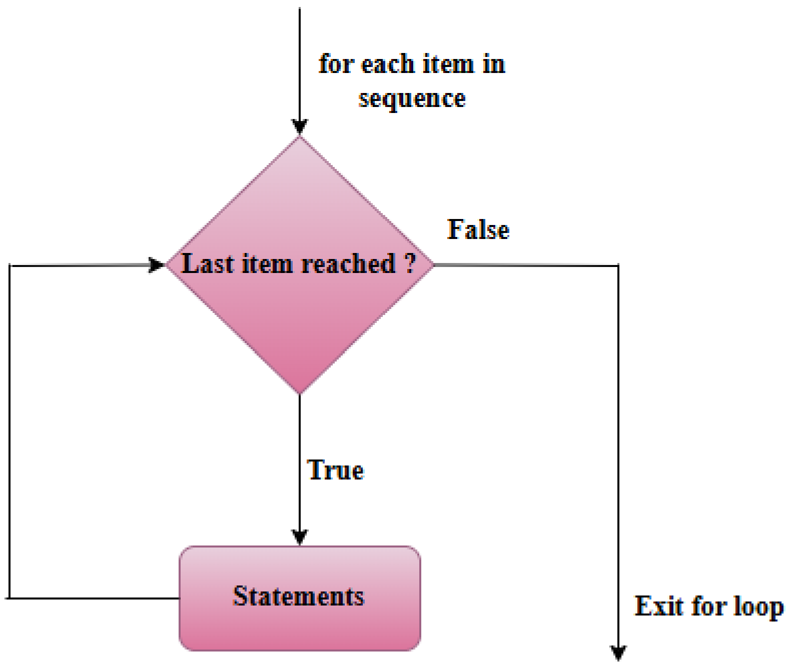


Fig 1.4.5 flowchart of for loop

Example:

```
items = [5, 10, 15, 20, 25]

sum = 0

for val in items:

        sum = sum+val

print("The sum is", sum)
```

Output:

```
The sum is 75
```

1.4.3.2 while loop

The while loop is used to repeat a set of statements as long as a specific condition remains true. It begins by checking the condition. If the condition is true, the loop's body (the block of code inside it) is executed. After completing one round of execution, the condition is checked again. If it is still true, the loop runs again. This process continues until the condition becomes false, at which point the loop stops. The while loop is especially useful when we do not know in advance how many times the loop should repeat. It allows the program to keep running a task until a certain situation or result is reached.

Syntax

```
Syntax:

for var in sequence:

        statement(s)
```

Flowchart:

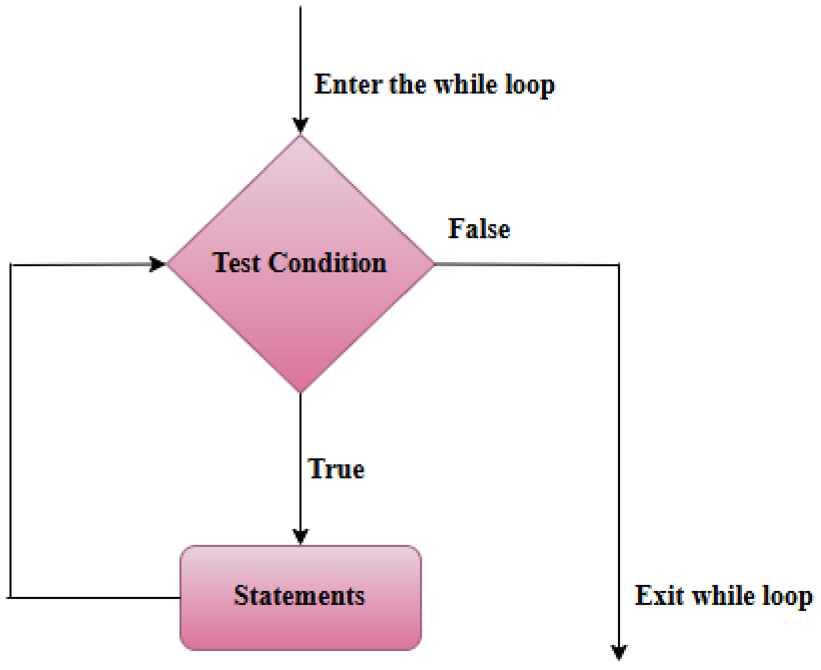


Fig 1.4.6 flowchart of while loop

Example

```
n = 10

sum = 0

i = 1

while i <= n:

   sum = sum + i

   i = i + 1

print("The sum is", sum)
```

Output:

```
The sum is 55
```

1.4.4 break Statement

The break statement is used to stop a loop immediately, even if its condition is still true or there are more items to process. It is commonly used when a certain condition is met and you want to exit the loop early. The break statement can be used inside both for and while loops.

Syntax:

```
loop:
    if condition:
        break

    # other statements
```

Example

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output:

```
0
1

2
```

1.4.5 continue Statement

The continue statement is used inside loops to skip the current iteration and proceed directly to the next one. When the Python interpreter encounters a continue statement inside a loop (either for or while), it immediately stops executing the remaining statements in the current iteration and moves to the next cycle of the loop. This is useful when specific values or cases need to be skipped during loop execution without stopping the entire loop.

Syntax:

```
for variable in sequence:
    if condition:
        continue

    # statements
```

Example:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Output:

```
0
1
3
4
```

Recap

- **Control Structures**: Direct the flow of program execution; divided into conditional statements and loops.
- **Conditional Statements**:
- if statement: Executes code when a condition is true.
- if-else statement: Executes one block if true, another if false.
- Nested if: Using one if or if-else inside another to handle multiple conditions.
- elif ladder: Checks multiple conditions in sequence for decision-making.
- **Loops**: Used to repeat code multiple times.
- for loop: Repeats for a fixed number of times or over items in a collection.
- while loop: Repeats as long as a condition remains true.
- **break Statement**: Immediately exits a loop when a condition is met.
- **continue Statement:** Skips the current iteration and continues with the next loop cycle.

Objective Type Questions

1. Which keyword is used to repeat a block of code a specific number of times?
2. What keyword is used to exit a loop early?
3. Which keyword skips the current iteration and moves to the next in a loop?
4. What is the keyword used to repeat a block of code as long as a condition is true?
5. What type of loop executes when the condition is true before each iteration?
6. What keyword is used to provide an alternative block of code in if statement?
7. What is the loop variable in for i in range(5)?
8. Which loop is preferred when the number of iterations is known?
9. Which loop is preferred when the number of iterations is not known?
10. How many times will a for loop run: for i in range(2, 6)?

Answers to Objective Type Questions

1. for
2. break
3. continue
4. while
5. while
6. else
7. i
8. for
9. while
10. 4

Assignments

1. Explain the working of if, elif, and else statements in Python with suitable examples.
2. Write a Python program to check whether a number entered by the user is even or odd using an if-else statement.
3. Describe the syntax and use of the for loop in Python. Write a program to print numbers from 1 to 10 using a for loop.
4. Write a Python program using a while loop to calculate the sum of natural numbers up to a given number n.
5. What is the difference between break and continue statements? Explain each with an example.

Reference

1. https://www.w3schools.com/python/

2. https://www.learnpython.org/

Suggested Reading

1. Brown, Martin C. *Python: The complete reference*. Osborne/McGraw-Hill, 2001.
2. Jose, Jeeva. *Taming Python by Programming*. KHANNA PUBLISHING HOUSE.
3. Lutz, Mark. *Learning python: Powerful object-oriented programming*. " O'Reilly Media, Inc.", 2013.

2

BLOCK

Data Structures in Python

Unit 1

List and Tuples

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- describe the need for using lists and tuples to store multiple values in a single variable efficiently.
- demonstrate the ability to create, access, modify, and delete elements in Python lists.
- differentiate between mutable (list) and immutable (tuple) data types and identify when to use each.
- apply indexing and slicing techniques to retrieve or manipulate specific portions of data from a list or tuple.

Prerequisites

In Python programming, variables can normally store only a single value at a time. When handling large amounts of related data, like storing names of 100 students or daily temperatures for a month, creating separate variables becomes inefficient. To solve this, Python provides data structures like lists and tuples that allow storing multiple values in a single variable. Lists are mutable, meaning the contents can be changed after creation, whereas tuples are immutable. This helps developers choose based on whether they want to protect or modify the data. Lists are useful when you need to update, insert, or remove items frequently. Tuples are preferred for fixed data like coordinates, configuration values, or constant lookup values. For example, in a school management system, student names can be stored in a list, while the fixed set of weekdays for class schedules can be stored in a tuple.

Key words

List, Tuple, Append, Insert, Pop, Slicing, Indexing, Immutable

Discussion

2.1.1 Python Lists

We have already discussed variable names. Only one data can be represented by a variable. For example, Student_name = “KKG”.

Sometimes we need to read, store, process, and finally, output many data, maybe dozens, perhaps even thousands of data. Do you create that many different variable names for each value? Then you will have to spend long hours writing statements as shown below.

```
    X = 20
    X1=100
    X2=30
  ………….
    Xn = 19
```

(Note: X is a variable name, it could be any name such as mark or age). Think of how easy and convenient it would be to use one variable that will store all these data as shown below.

X = [20,100,30 … 19]

Using List is one of the solutions. We can use the list to store more than one data under one name. A list is a collection of data, similar to an array in many other programming languages. Lists might contain items of different types, but usually, the items will be the same type.

Since we can add and remove items, we say that a list is a mutable data type in which data can be altered.

The list can be used to store multiple items or values or data in a single variable name. for example x= [“KKG”, “pen”,”beach”]. x represents a list. If we write x = “KKG”, x is a string variable name that stores only one data.

The list is defined in Python as a list of comma-separated values (items) between [] square brackets.

The items in the list can be accessed using the index operator [].

Age = [2,4,1,10,5]. This List has 5 items. The first value is saved as Age[0] = 2, second item Age[1] = 4. The third value is in Age [2], fourth value in Age[3] and fifth value in Age[4]. A list with n items, the index will start from 0 and the last index is n-1

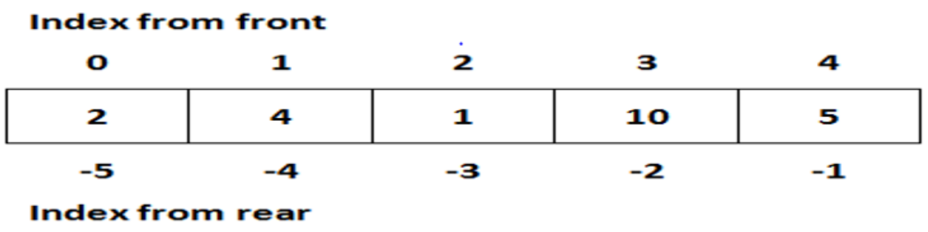


Fig 1.3.3

The value or expression inside the brackets represents the index. The index can be an integer or integer expression. Remember that the indices start at 0: In Age[0], 0 is the index

Example of integer expression as index:

```
I = 3
```

Age[I + 1] represents Age[4]

Example: Student_List = [“John”, “KKG”,”Jane”]

The following list is an example of a list that contains a string, a float, an integer, and another list:

```
L = [‘Covid’, 2.0, 5, [10, 20]]
```

A list that contains no items is called an empty list.

Example: Lis = [].

In the following activities guess the output first, then run the code and check the result.

Activity 1: Write and Run the below code and see the result

```
lis= []

print(lis)
```

Output : []

Activity 2: Run the below code and observe the result.

```
Student_List= [“John”, “KKG”, “Jane”]

print(Student_List)
```

Output: [‘John’, ‘KKG’, ‘Jane’]

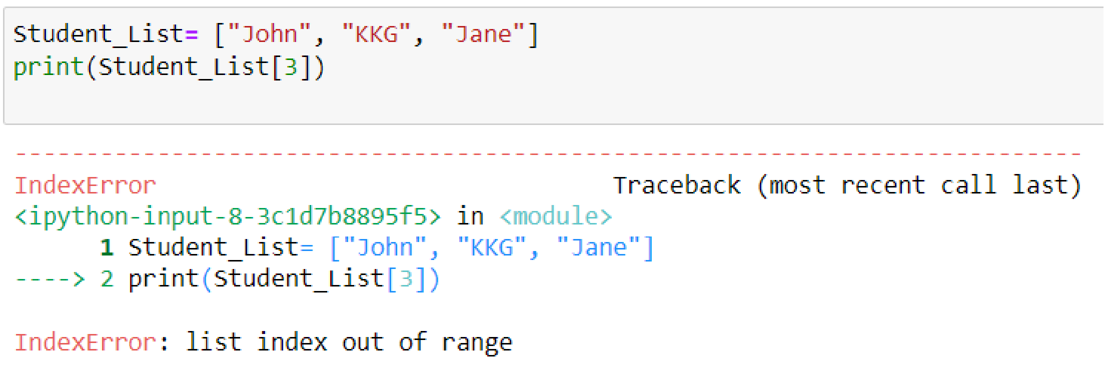
Activity 3: Write and Run the below code and check the result

```
Student_List= [“John”, “KKG”, “Jane”]

print(Student_List[0])
```

Output: John

If you try to read an element that does not exist, you will get an IndexError.

Run the below code and see the result



```
Student_List= ["John", "KKG", "Jane"]
print(Student_List[3])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-8-3c1d7b8895f5> in <module>
      1 Student_List= ["John", "KKG", "Jane"]
----> 2 print(Student_List[3])

IndexError: list index out of range
```

Fig 1.3.4

The list named Student_List shown above has three items. The first item index is 0, the second index is 1 and the third index is 2. If we try to print the Student_List with index 3 will give an error message.

Python lists are built-in, dynamically resizing arrays that automatically grow or shrink as needed. They can hold any kind of object, including other lists - because they store references in contiguous memory, whereas the actual objects might reside elsewhere. This design allows lists to contain mixed data types and duplicate items. Lists are mutable, meaning elements can be modified, replaced, or removed in place. They are ordered, preserving the insertion sequence, and elements are accessed via zero-based indexing, allowing direct access by position.

Example

a = [10, 20, “GfG”, 40, True]

print(a)

print(a[0], a[1], a[2], a[3], a[4])

print(type(a[2]), type(a[4]))

This example shows a list containing integers, a string, and a boolean; accessing elements by index retrieves and confirms their types.

Memory Model: Lists Store References, Not Values

A list object doesn’t hold the actual items directly; instead, it holds pointers to objects stored separately in memory. Python creates individual objects for 10, “GfG”, True, etc., and the list contains references to these locations. Modifying one element does not change others, although if the element refers to a mutable object, that object itself can be mutated.

2.1.1.1 Creating Lists

a. Using square brackets:

```
a = [1, 2, 3, 4, 5]
b = [‘apple’, ‘banana’, ‘cherry’]
c = [1, ‘hello’, 3.14, True]
```

b. Using the list() constructor with iterables:

```
a = list((1, 2, 3, ‘apple’, 4.5))
```

c. With repeated elements using the multiplication operator:

```
a = [2] * 5      # [2, 2, 2, 2, 2]
b = [0] * 7      # [0, 0, 0, 0, 0, 0, 0]
```

2.1.1.2 Accessing Elements

You can retrieve elements using both positive and negative indices. The first element has index 0 and the last element can be accessed with index −1.

```
a = [10, 20, 30, 40, 50]
print(a[0])   # 10
print(a[-1])  # 50
```

Modifying a List

Add elements:

- append(x) → adds one element at the end
- insert(idx, x) → places an element at a specific position
- extend(iterable) → adds multiple elements from an iterable

Update an element via index:

```
a[1] = 25
```

Remove elements:

remove(val) → removes first occurrence of val

pop([idx]) → removes and returns element at idx (defaults to last)

del a[idx] → deletes element at given index

2.1.1.3 Iterating Through a List

Using a simple for loop:

```
for item in ['apple', 'banana', 'cherry']:
    print(item)
```

Nested Lists

Lists can contain other lists, making them useful for representing nested structures like matrices. Access nested elements via chained indexing:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(matrix[1][2])

Outputs 6
```

List Comprehensions

A streamlined way to generate lists in a single line using an iterable and expression:

```
squares = [x**2 for x in range(1, 6)]

print(squares)

Outputs [1, 4, 9, 16, 25]
```

Theoretical Insight

Python lists are implemented as dynamic arrays, similar to C++ vectors or Java ArrayLists. They support O(1) time for indexing and amortized O(1) for append operations, thanks to occasional over-allocation strategies. Lists are best suited for collections where frequent random-access, addition, or deletion at end operations are needed. However, inserts or deletions in the middle or at the front can be slower (O(n)), because elements have to be shifted.

2.1.2 Tuples in python

A **tuple** allows you to group multiple values into a single variable.

It is one of the four primary built-in data structures in Python designed for storing collections. The other three are **lists**, **sets**, and **dictionaries**, each serving different purposes and having unique characteristics.

Tuples are **ordered** collections, meaning items maintain their insertion order, and they are **immutable**, meaning their contents cannot be modified after creation.

Tuples are defined using **parentheses**.

Example:

```
thistuple = ("apple", "banana", "cherry")

print(thistuple)
```

Output:

```
('apple', 'banana', 'cherry')
```

This code creates a tuple with three string elements and prints its contents.

2.1.2.1 Tuple Items in Python

Tuple elements are **indexed**, **ordered**, and **immutable**, and they also support **duplicate values**.

Each item in a tuple can be accessed using its position, starting from index 0 for the first item, 1 for the second, and so on.

Ordered Nature

Tuples maintain the sequence in which elements are added. Once defined, the position of elements remains fixed.

Immutable

Tuples cannot be modified after they are created. This means you cannot update, insert, or delete any elements once the tuple is formed.

Supports Duplicates

Because tuple elements are indexed, identical values can appear more than once within the same tuple.

Example:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")

print(thistuple)
```

Output:

```
('apple', 'banana', 'cherry', 'apple', 'cherry')
```

This example shows a tuple that includes repeated items. The values "apple" and "cherry" appear more than once, which is perfectly valid in a tuple.

2.1.2.2 Length of a Tuple

To find out the total number of elements present in a tuple, you can use Python's built-in len() function.

Example:

```
thistuple = ("apple", "banana", "cherry")

print(len(thistuple))
```

This code prints the count of items in the tuple thistuple, which in this case is **3**.

2.1.2.3 Creating a Tuple with a Single Element

When defining a tuple that contains just one item, it's essential to include a comma after the item. Without the comma, Python will treat it as a regular value (like a string) instead of a tuple.

Example:

```
# This is a tuple with one item

thistuple = ("apple",)

print(type(thistuple))

# Output: <class 'tuple'>

# This is just a string, not a tuple

thistuple = ("apple")

print(type(thistuple))  # Output: <class 'str'>
```

Note: The trailing comma is what makes it a tuple, even if there is only one item.

2.1.2.4 Tuple Items – Data Types

Items stored in a tuple can be of **any data type**.

Example:

You can create tuples containing:

- Strings
- Integers
- Boolean values

```
# Tuple with string values

tuple1 = ("apple", "banana", "cherry")

# Tuple with integer values

tuple2 = (1, 5, 7, 9, 3)

# Tuple with boolean values

tuple3 = (True, False, False)
```

Note: Tuples are versatile and can hold any type of data, including combinations of different types.

A tuple can contain different data types:

A tuple with strings, integers and boolean values:

tuple1 = (“abc”, 34, True, 40, “male”)

2.1.2.5 type() Function

In Python, when you use the type() function on a tuple, it shows that the object is of the tuple data type.

Example Output:

<class ‘tuple’>

This confirms that the variable is recognized by Python as a tuple object.

2.1.2.6 The tuple() Constructor

You can also create a tuple using the built-in tuple() function.

Example

Using the tuple() constructor to create a tuple:

thistuple = tuple((“apple”, “banana”, “cherry”)) # notice the use of double parentheses

print(thistuple)

This method is useful when converting other iterable objects like lists or strings into a tuple.

Python Collections

Python provides four main types of collections for storing groups of items:

- **List**: An ordered and mutable collection that allows duplicate values.
- **Tuple**: An ordered and immutable collection that also permits duplicates.
- **Set**: An unordered, unindexed, and mostly immutable collection that does not allow duplicates.
- **Dictionary**: An ordered and mutable collection of key-value pairs with unique keys (no duplicates).

Each collection type serves different purposes depending on how you want to organize and manage data.

2.1.3 List Indexing and Slicing

Lists can be indexed and sliced. Indexing returns the item as shown below.

Example: Student_List = [“John”, “KKG”,”Jane”]

Student_List[0] returns the item John

Student_List[1] returns the item KKG

Sometimes we need to access or process part of the list. For example, display the first 100 student names from a list with a total of 1000 students’ names. Slice operations return a new list containing the requested elements.

Syntax: List[start : end : step]

The list is the name of List

Start represents the starting index

End represents the ending index

Step represents the increment or decrement value for the index.

Example:

```
Student_List= ["John", "KKG", "Jane"]
print(Student_List[0:2:1])
```

The output of the above code is

```
['John', 'KKG']
```

The start index in the above example is zero, the end index is 2 and the step is 1. The slicing will start from the index zero and output the name John, then the index will be incremented to one.(The step is 1) and output the name KKG, then the index will be incremented to 2, but the item will not be returned. Remember the index start from zero and end at n-1

Python List Slicing

Python list slicing is a core concept that enables direct access to specific parts of a list using slicing syntax. It allows us to extract sublists using both positive and negative indices with concise code.

Example: Extract elements from index 1 to 4 (exclusive)

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a[1:4])
```

Output

```
[2, 3, 4]
```

Syntax

list_name[start : end : step]

- **start** (optional): inclusive starting index (defaults to 0)
- **end** (optional): exclusive ending index (defaults to list length)
- **step** (optional): interval between elements (defaults to 1)

Examples

1. All items

print(a[:])   # or a[::]

Returns the full list.

2. Before or after a position

/////'////////

print(a[2:])   # from index 2 to end

print(a[:3])   # from start up to index 3 (exclusive)

3. Between two indices

print(a[1:4]) # elements 2 through 4

4. With step intervals

print(a[::2])   # every 2nd element

print(a[1:8:3])   # elements at steps of 3 starting index 1

5. Out-of-bound slicing

print(a[7:15]) # returns [8, 9], no error

Gracefully handles indices beyond list range.

6. Negative indexing

print(a[-2:])   # [8, 9]

print(a[:-3])   # all except last three

print(a[-4:-1])   # slice from -4 to -1

print(a[-8:-1:2]) # spaced slicing with negative indices

7. Reversing via slicing

print(a[::-1])

A neat and efficient way to create a reversed copy using a negative step

A Bit of Theory

In Python slicing operates via a slice(start, stop, step) object, which the Python interpreter uses to copy list segments. If the start is greater than end, or incompatible with step, slicing returns an empty list rather than causing an error. This design promotes robustness and avoids unnecessary exceptions.

2.1.4 Sequence

A sequence type is a type of data such as a list in Python that can store more than one value. A sequence type data structure can store zero(for example, an empty list), one or more values. There are 3 more types of sequence data type that function differently than the list.

- Dictionary
- Tuple
- Sets

2.1.5 Differences between List, Tuple, Dictionary, and set

Table 1.3.1

Property	List	Tuple	Dictionary	Set
Ordered	Yes	Yes	From version 3.7	No
Indexed	Yes	Yes	Yes	No
Key and Value Pair	No	No	Yes	No
Bracket type	[]	( )	{ }	{ }
Changeable(add/Remove values)	Yes	No	Yes	Yes
Allow duplicate values	Yes	Yes	No	No

List, Tuple, Dictionary, and set are used to store multiple values using a single variable name

2.1.6 Data types and examples

Table 1.3.2

Data type	Example	Result	Keyword
Integer	Number_of_Children = 2 print (Number_of_Children)	2	int

Float	mark = 89.5
	print(mark)
	print(type(mark))
		89.5
		<class 'float'>
			float

	x =6.62607E-34
	print(x)
	print(type(x))
		6.62607e-34
		<class 'float'>
			float

String	Student_name = “KKG”	KKG	str

List	x= ["john", "pen","beach"]
	print(x)
	print(type(x))
		['john', 'pen', 'beach']
		<class 'list'>
			list

Tuples	x= ("john", "pen", "beach")
	print(x)
	print(type(x))
		('john', 'pen', 'beach')
		<class 'tuple'>
			tuple

Dictionary	x ={"Mark":20, "Name": "John", "year": 2021}
	print(x)
	print(type(x))
		'Mark': 20, 'Name': 'John', 'year': 2021}
		<class 'dict'>
			dict

Set	x ={20, "John", 2021}
	print(x)
	print(type(x))
		{20, 2021, 'John'}
		<class 'set'>
			set

Boolean	x =True
	print(x)
	print(type(x))
		True
		<class 'bool'>
			bool

Recap

- A list allows storing multiple items in a single variable using square brackets [].
- Lists are mutable, meaning elements can be changed after creation.
- Items in a list can be of mixed data types (e.g., integers, strings, booleans, even other lists).
- Lists maintain insertion order and support duplicate values.
- You can access list elements using zero-based indexing (e.g., x[0] for the first item).
- Negative indexing allows accessing items from the end (e.g., x[-1] for the last item).
- An empty list is created using [] or list().
- Lists can be created using the list() constructor from any iterable (e.g., list((1, 2, 3))).

- Use append() to add a single element to the end of the list.
- Use insert(index, value) to add an element at a specific position.
- Use extend(iterable) to add multiple elements from another iterable.
- Use pop(index) to remove and return an element at a specific index (default is the last).
- Use remove(value) to delete the first occurrence of a specific value.
- Use del list[index] to delete an element by its index.
- Lists can be iterated using a for loop.
- Nested lists are supported, and you can access inner elements using double indexing (e.g., matrix[1][2]).
- List slicing allows extracting sublists using [start:end:step] syntax.
- Slicing supports positive and negative indices, and doesn't raise errors for out-of-bound ranges.
- List comprehension provides a compact syntax for creating new lists using an expression and iterable.
- Python internally store references to objects, not the actual values.

Objective Type Questions

1. What data type allows storing multiple values under a single variable name in Python?
2. Which brackets are used to define a list in Python?
3. What is the index of the first item in a Python list?
4. What is the result of accessing an index beyond the list's length?
5. Which list method is used to add an element at the end of a list?
6. What type of error is raised when trying to access a non-existing list index?
7. Which list method removes and returns an item at a given index?
8. A list that contains no elements is called a __________ list.
9. Which keyword confirms that a list is a mutable data type?
10. How do you access the last element in a list using negative indexing?

11. What is the output of print([1,2,3,4][::-1])?
12. Which method is used to combine two lists?
13. Which function returns the total number of items in a list?
14. What is the time complexity of appending an item to a list?
15. What is the keyword used to define a list using the constructor?

Answers to Objective Type Questions

1. List
2. Square
3. Zero
4. IndexError
5. append
6. IndexError
7. pop
8. Empty
9. Mutable
10. [-1]
11. [4, 3, 2, 1]
12. extend
13. len
14. O(1)
15. list

Assignments

1. Explain the concept of Python Lists. Describe how lists are created, accessed, and modified. Include examples demonstrating creation, indexing, slicing, and mutability.
2. Differentiate between Lists and Tuples in Python. Provide a comparison based on mutability, syntax, use-cases, and performance, along with examples.
3. Write a Python program that performs the following operations on a list:

   Create a list with at least 5 integers

   Append a new element

   Insert an element at index 2

   Remove the last element

   Print the final list
4. What is list slicing in Python? Explain the slicing syntax with positive and negative indices. Include at least three examples showing different slicing patterns.
5. Describe the memory model of Python Lists. How do Python lists store elements internally? Explain how mutability and referencing work with an example that includes nested lists.

Reference

1. Python online documents. https://docs.python.org/

Suggested Reading

1. A Beginner's Guide To Learn Python In 7 Day, Author: Ramsey Hamilton
2. Python Programming for Beginners: Learn The Basics Of Python Programming (Python Crash Course, Programming for Dummies). Author: James Tudor
3. https://www.python.org/about/gettingstarted/

Unit 2
Dictionaries and Sets

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- recall the definition of a Python dictionary.
- list the key characteristics of Python sets.
- identify methods to add elements to a set in Python.
- recognize the built-in functions used for dictionary item access.
- name the four primary set operations in Python

Prerequisites

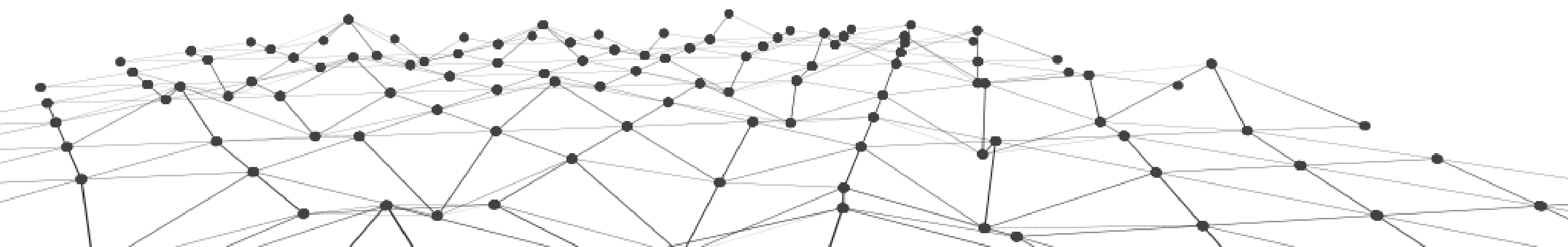
Imagine you are developing a contact management system where you need to store and organize information about people's phone numbers, email addresses, and names. You want to make sure each contact is unique and easily searchable. At the same time, you want to quickly find common contacts shared between your friends or identify contacts exclusive to your list.

To do this efficiently, you use **dictionaries** to store contact details as key-value pairs where each person's name is a unique key linked to their contact info. Meanwhile, you use **sets** to manage groups of contacts, so you can quickly perform operations like finding contacts you and your friend both have or all contacts combined.

This unit will teach you how to use these powerful data structures, their key features, and how to apply set operations and dictionary manipulations to solve real-world problems like this effectively.

Key words

Key-value pairs, Mutable, Union, Intersection, Symmetric Difference



Discussion

In Python, **sets** and **dictionaries** are powerful built-in data structures that provide efficient ways to store and manage data. Both are based on **hash tables,** making them optimized for fast data access, insertion, and deletion. While they share similarities in performance and underlying implementation, they serve different purposes: sets are used to store **unordered collections of unique elements**, whereas dictionaries store **key-value pairs**, allowing for quick lookup of values based on unique keys. These structures are essential tools for writing clean, efficient, and expressive Python code.

2.2.1 Dictionary

A Python dictionary is a data structure used to store data in key-value pairs. The keys must be unique and immutable, while the values can be of any data type and may be duplicated.

In the dictionary below, data is organized using key-value pairs, allowing for easy and efficient value retrieval

```
d = {1: 'How', 2: 'Are', 3: 'You'}

print(d)
```

Output

```
{1: 'How', 2: 'Are', 3: 'You'}
```

2.2.1.1 Key Characteristics of Dictionary

1. **Unordered (prior to Python 3.7)** – In versions before Python 3.7, dictionaries did not preserve the order in which items were inserted. However, starting from Python 3.7, maintaining insertion order became a guaranteed feature of the language.

2. **Mutable** – Dictionaries can be modified after creation; you can add new key-value pairs, change existing ones, or delete items.

3. **Key-Based Access** – Instead of using numerical indexes like lists, dictionaries use keys to retrieve their corresponding values.

4. **Unique Keys** – Every key in a dictionary must be distinct. If a duplicate key is used during assignment, the previous value is overwritten.

5. **Mixed Data Types** – Both keys and values in a dictionary can be of various data types, allowing for flexible usage.

2.2.2 Key-Value Pairs

In Python, dictionaries store information as key-value pairs. Each key serves as a unique identifier for its value, enabling quick access or modification of the value through the key.

Syntax: **key: value**

Multiple pairs are separated by commas and enclosed within curly braces {}.

- Keys must be unique and of an immutable type, such as strings, numbers, or tuples.
- Values can be of any data type and do not need to be unique.

```
person = {
    “name”: “John”,
    “age”: 25,
    “city”: “New York”,
    “age”: 30
}
print(person)
```

Output:

```
{‘name’: ‘John’, ‘age’: 30, ‘city’: ‘New York’}
```

If the same key appears more than once, the last value assigned to it will be retained.

2.2.3 Dictionary Operations

Dictionary operations in Python refer to the common actions you can perform on dictionaries like adding, accessing, updating, deleting items, and more.

2.2.3.1 Accessing Dictionary Items

We can access a value from a dictionary by using the **key** within square brackets or get() method.

```
d = { “name”: “Prajjwal”, 1: “Python”, (1, 2): [1, 2, 4] }
print(d[“name”])            # Using square brackets
print(d.get(“name”))        # Using the get() method
```

Output

Prajjwal

Prajjwal

Both methods return the value associated with the key “name ”.

2.2.3.2 Adding and Updating Dictionary Items

We can **add new key-value pairs** or **modify existing ones** in a dictionary using assignment

```
d = {1: 'Python', 2: 'For', 3: 'All'}
d["age"] = 22                    # Adding a new key-value entry
d[1] = "Python dict"             # Modifying the value of an existing key
print(d)
```

Output

```
{1: 'Python dict', 2: 'For', 3: 'All', 'age': 22}
```

In this example, a new key "age" is added with the value 22, and the value for the key 1 is updated to "Python dict".

2.2.3.3 Deleting Items from a Dictionary

To remove elements from a dictionary we can use the following methods

- **del:** Deletes a specific key and its associated value.
- **pop():** Removes a key and returns the corresponding value.
- **clear():** Removes all items, leaving the dictionary empty.
- **popitem():** Deletes and returns the most recently added key-value pair.

```
d = {1: 'Python', 2: 'For', 3: 'All', 'age':22}
del d["age"]                  # Using del to remove an item
print(d)
# Using pop() to remove an item and return the value
val = d.pop(1)
print(val)
# Using popitem to removes and returns the last key-value pair
key, val = d.popitem()
print(f"Key: {key}, Value: {val}")
# Clear all items from the dictionary
d.clear()
print(d)
```

Output

```
{1: ‘Python’, 2: ‘For’, 3: ‘All’}

Python

Key: 3, Value: All

{}
```

2.2.3.4 Iterating through a Dictionary

You can loop through a dictionary using a for loop to access

- Keys with the keys() method
- Values with the values() method
- Both keys and values using the items() method

```
d = {1: ‘python’, 2: ‘For’, ‘age’:22}

# Iterate over keys

for key in d:

   print(key)

# Iterate over values

for value in d.values():

   print(value)

# Iterate over key-value pairs

for key, value in d.items():

   print(f’{key}: {value}”)
```

Output

```
1

2

age

Python

For

22

1: Python

2: For

age: 22
```

2.2.4 Sets

In Python, a **set** is an **unordered collection of unique elements**. Unlike lists or tuples, sets do not allow duplicate values i.e.; each element in a set must be distinct. Sets are **mutable**, which means you can add or remove elements after the set has been created.

Sets are defined using **curly braces** `{}` or the built-in `set()` **function.** They are especially useful for **membership testing**, **removing duplicates** from sequences, and performing **common set operations** such as **union, intersection,** and **difference**.

Conceptually, a set represents a collection of distinct objects. It is used to group items and examine their properties and relationships. The elements in a set are referred to as **members** or **elements** of the set.

2.2.4.1 Characteristics of sets in Python

1. Unordered

- The elements in a set are **not stored in any specific order**.
- When printed or iterated, their order may differ each time.

2. Unindexed

- Sets do **not support indexing or slicing.**
- You **cannot access elements by position** like `s[0]`.

3. No Duplicate Elements

- A set **automatically removes duplicate values.**

4. Mutable

- You can **add or remove elements** from a set after its creation.
- Methods like `add()`, `remove()`, and `update()` modify the set.

2.2.4.2 Creating Sets

i. Using Curly braces

The easiest and fastest way to define a set in Python is by enclosing the elements within curly braces.

```
set1 = {1, 2, 3, 4}
print(set1)
```

Output

```
{1, 2, 3, 4}
```

ii. Using the set() function

In Python, sets can be created either by using the built-in `set()` function with an iterable or by placing elements inside curly braces `{}`, with each item separated by a comma.

```
my set = set([1,2,3,4,5])

print (my set)
```

Output

```
{1, 2, 3, 4, 5}
```

2.2.4.3 Adding Elements to a Set

Items can be added to a set using the `add()` and `update()` methods. The `add()` method is used to insert a single element, while the `update()` method allows you to add multiple elements at once.

```
set1 = {1, 2, 3}            # Creating a set

set1.add(4)                 # Add one item

set1.update([5, 6])         # Add multiple items

print(set1)
```

Output

```
{1, 2, 3, 4, 5, 6}
```

2.2.4.4 Accessing a Set

Since sets are unindexed and do not support element access by position, we use loops to go through their items. Additionally, the `in` keyword, a membership operator, can be used to check whether a specific element is present in the set.

```
set1 = set([“Python”, “For”, “All.”])

# Accessing element using For loop

for i in set1:

   print(i, end=” “)

# Checking the element using in keyword

print(“Python” in set1)
```

Output

```
All. Python For True
```

The order of the first three words may vary because **sets are unordered** collections in Python. The presence check (`“Python” in set1`) will always return `True`.

2.2.4.5 Removing an Element

In Python, elements can be removed from a set using different methods, each with its own behavior:

- remove() and discard() methods can be used to delete a specific element from the set.
- pop() removes and returns an arbitrary element since sets are unordered.
- clear() deletes all elements from the set, leaving it empty.

i. Using remove() or discard() Method

The `remove()` method deletes a specific element from a set, but if that element is not present, it raises a `KeyError`. In contrast, the `discard()` method also removes a specified element, but it does **not** throw an error if the element is missing from the set.

```
# Using Remove Method

set1 = {1, 2, 3, 4, 5}

set1.remove(3)

print(set1)

# Attempting to remove an element that does not exist

try:

    set1.remove(10)

except KeyError as e:

    print("Error:", e)

# Using discard() Method

set1.discard(4)

print(set1)

# Attempting to discard an element that does not exist

set1.discard(10)  # No error raised

print(set1)
```

Output

```
{1, 2, 4, 5}

Error: 10

{1, 2, 5}

{1, 2, 5}
```

ii. Using pop() Method

The `pop()` method deletes and returns a random element from the set, meaning the specific item removed is unpredictable. If the set is empty, using `pop()` will result in a `KeyError`.

```
set1 = {1, 2, 3, 4, 5}
val = set1.pop()
print(val)
print(set1)
# Using pop on an empty set
set1.clear()  # Clear the set to make it empty
try:
    set1.pop()
except KeyError as e:
    print("Error:", e)
```

Output

```
1
{2, 3, 4, 5}
Error: 'pop from an empty set'
```

iii. Using clear() Method

The `clear()` method deletes every element in the set, resulting in an empty set.

```
set1 = {1, 2, 3, 4, 5}
set1.clear()
print(set1)
```

Output

```
set()
```

2.2.5 Set Operations

Python set operations like union, intersection, difference, and symmetric difference are essential tools for working with unique collections of elements. These built-in operations allow you to combine, compare, and analyze sets efficiently in Python.

2.2.5.1 Union

It merges elements from both sets, and can be performed using the `union()` method or the `|` operator.

```
a = {1, 2, 3}
b = {3, 4, 5}
```

```
print(a | b)              # Output: {1, 2, 3, 4, 5}
print(a.union(b))         # Output: {1, 2, 3, 4, 5}
```

2.2.5.2 Intersection

It is used to find shared elements between sets using the `intersection()` method or the `&` operator.

```
a = {1, 2, 3}
b = {2, 3, 4}
print(a & b)                   # Output: {2, 3}
print(a.intersection(b))      # Output: {2, 3}
```

2.2.5.3 Difference

It retrieves elements that exist in one set but not in the other, using the `difference()` method or the - operator.

```
a = {1, 2, 3}
b = {2, 3, 4}
print(a - b)                  # Output: {1}
print(a.difference(b))       # Output: {1}
```

2.2.5.4 Symmetric Difference

It returns elements that are present in either of the sets but not in both, using the `symmetric_difference()` method or the `^` operator.

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a ^ b)                              # Output: {1, 2, 4, 5}
print(a.symmetric_difference(b))         # Output: {1, 2, 4, 5}
```

Recap

- ♦ Python sets and dictionaries are built-in data structures optimized for fast access, insertion, and deletion using hash tables.
- ♦ Dictionaries store data as key-value pairs with unique, immutable keys and values of any type.
- ♦ Dictionaries support adding, updating, and deleting items.
- ♦ Values in dictionaries can be accessed using keys with square brackets or the get() method.
- ♦ Items in dictionaries can be deleted using del, pop(), clear(), or popitem().
- ♦ Sets are unordered collections of unique elements and do not allow duplicates.
- ♦ Sets are mutable and can be created using curly braces {} or the set() function.
- ♦ Elements can be added to a set using the add() method (single item) or update() method (multiple items).
- ♦ Sets do not support indexing or slicing; elements are accessed by looping or using the in keyword to check membership.
- ♦ Elements can be removed from a set using remove() (raises error if element not found), discard() (no error if element missing), pop() (removes and returns an arbitrary element), and clear() (removes all elements).
- ♦ Set operations include:
    - Union (union() method or | operator) merges elements from both sets.
    - Intersection (intersection() method or & operator) finds common elements.
    - Difference (difference() method or - operator) gets elements in one set but not the other.
    - Symmetric difference (symmetric_difference() method or ^ operator) returns elements in either set but not both.

Objective Type Questions

1. What data structure in Python stores key-value pairs?
2. What keyword is used to define a set in Python?
3. Which operator is used for the union of two sets?

4. Which method removes all items from a dictionary?
5. Which data structure stores only unique elements?
6. What operator is used for checking membership in a set?
7. Which method removes a random element from a set?
8. Which dictionary method returns the value for a given key safely?
9. Which method returns shared elements between two sets?
10. Which method adds a single element to a set?

Answers to Objective Type Questions

1. Dictionary
2. set
3. |
4. clear()
5. Set
6. in
7. pop()
8. get()
9. intersection()
10. add()

Assignments

1. Describe the key features that differentiate Python sets from dictionaries.
2. Write a Python program to demonstrate all four major set operations: union, intersection, difference, and symmetric difference.
3. Explain the behavior and usage of key-value pairs in Python dictionaries with suitable examples.

4. Create a set from a list containing duplicate elements and display the result to show how duplicates are handled.

5. Develop a dictionary to store and manage contact details (name and phone number) and perform addition, update, deletion, and retrieval operations.

Reference

1. Beazley, D., & Jones, B. (2013). *Python cookbook* (3rd ed.). O'Reilly Media.

2. Hetland, M. L. (2005). *Python programming: An introduction to computer science*. Franklin, Beedle & Associates Inc.

3. Martelli, A., Ravenscroft, A., & Ascher, D. (2006). *Python in a nutshell*. O'Reilly Media.

4. Pilgrim, M. (2009). *Dive into Python 3*. Apress.

5. Grinberg, M. (2018). *Flask web development: Developing web applications with Python*. O'Reilly Media.

Suggested Reading

1. https://docs.python.org/3/tutorial/datastructures.html

2. Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. Python Software Foundation.

3. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

4. Sweigart, A. (2015). *Automate the boring stuff with Python*. No Starch Press.

5. Downey, A. (2015). *Think Python: How to think like a computer scientist* (2nd ed.). Green Tea Press

Unit 3

Strings and String Manipulation

Learning Outcomes

Upon completion of this unit, the learner will be able to

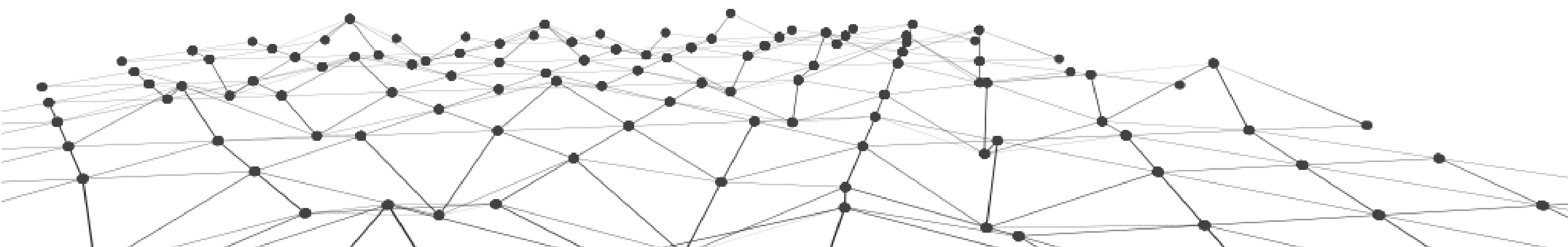
- explain the concept of strings and how to declare them using quotes in Python.
- use escape sequences to properly format output text in a Python program.
- apply various string methods like replace(), split(), join(), find() to manipulate text.
- perform string comparisons using relational operators and understand lexicographical order.
- utilize indexing and slicing to access and extract parts of strings efficiently.

Prerequisites

Before diving into string operations in Python, it's important to understand **why** studying strings is essential. In the digital world, **textual data** is everywhere such as names, messages, emails, addresses, websites, search queries, and even programming code itself. All of this data is handled as **strings** in most programming languages, including Python. Learning how to work with strings is fundamental to being able to process, analyze, and manipulate real-world data.

Consider a real-life example: Imagine you're creating a messaging app. Every time a user sends or receives a message, you're dealing with strings. You might need to check if a message starts with a certain keyword (e.g., "URGENT"), convert everything to lowercase for uniformity, or even search for specific phrases within messages. Without understanding how to use string methods, slicing, or comparisons, these tasks would be impossible to perform effectively.

Understanding strings also lays the foundation for more advanced topics like **data processing**, **file handling**, **web development**, and **natural language processing (NLP)**. Therefore, a strong grasp of strings and their operations is a crucial skill for any aspiring Python programmer.



Key words

String, Slicing, Indexing, Formatting, Immutable, Method, Concatenation, Join, Split, Escape Sequence

Discussion

A string in Python is a sequence of characters enclosed within either single quotes ' ' or double quotes " ". Strings are used to represent text-based data such as names, addresses, and sentences. Python strings are immutable, which means once created, the contents of a string cannot be changed.

Examples:

name = “Alice”

message = ‘Welcome to Python!’

Think of a string like a train made up of boxcars (characters) arranged in a specific order. Once the train (string) is formed, you cannot modify a single boxcar directly, you’d need to create a new train instead.

Multi-line Strings: If we need a string to span multiple lines then we can use **triple quotes (“‘ or ”””)**

You can create strings in various ways:

- **Single Quotes:** ‘Hello’
- **Double Quotes:** “World”
- **Triple Quotes:** “‘ This is a multiline string’’’ or ““““ This also works””””

s = ““““I am Learning

Python String for fun””””

print(s)

s = “‘I’m priya’’’

print(s)

#Output

I am Learning

Python String for fun

I’m priya

2.3.1 String Methods

Python strings come with many built-in methods for manipulation.

1. lower() and upper()

Convert to lowercase or uppercase:

```
   name = "Alice"
   print(name.lower())  # alice
   print(name.upper())  # ALICE
```

2. strip()

Removes leading/trailing spaces:

```
txt = " hello"
print(txt.strip())  # "hello"
```

3. replace()

Replaces a substring with another:

```
text = "Python is easy"
print(text.replace("easy", "fun")) #Python is fun
```

4. split()

Splits the string into a list:

```
sentence = "Python is awesome"
print(sentence.split())  # ['Python', 'is', 'awesome']
```

5. join()

Joins elements of a list into a string:

```
words = ["Python", "is", "fun"]
print(" ".join(words))  #Python is fun
```

6. find() and index()

find()

Purpose: Returns the index of the first occurrence of the substring.

Return Value: Returns -1 if the substring is not found.

Safe to use when you're unsure if the substring exists.

index()

Purpose: Also returns the index of the first occurrence of the substring.

Return Value: Raises a ValueError if the substring is not found.

Use it only if you're sure the substring is present.

```
print("hello".find("e"))  # 1
print("hello".index("e"))  # 1
print("hello".find("a"))   # Output: -1 (not found)
```

7. startswith() and endswith()

startswith()

Purpose: Checks if a string starts with the specified substring.

Returns: True if it does, False otherwise.

endswith()

Purpose: Checks if a string **ends** with the specified substring.

Returns: True if it does, False otherwise.

```
text = "programming"
print(text.startswith("pro"))  # True
print(text.endswith("ing"))   # True
```

2.3.1.1 String Operations

1. Escape Characters in Strings

Some characters can't be typed directly, so escape sequences are used:

Table 2.3.1 escape sequence

Escape Sequence	Description
\n	New line
\t	Tab
\"	Double quote
\\	Backslash

Example:

```
print("Line1\nLine2")
```

Output

```
Line1
Line2
```

2. String Immutability

Once a string is created, its characters cannot be changed. Any modification results in a new string.

Example:

```
name = “John”

# name[0] = “P”  # Error: strings are immutable

name = “Paul”  # Assigns a new string instead

print(name)    # Output: Paul
```

3. Looping Through Strings

In Python, a string is a **sequence of characters**, so you can iterate over it using a **for** loop.

```
for char in “Python”:

    print(char)
```

#Output

```
  P

  y

  t

  h

  o

  n
```

“Python” is a string made up of 6 characters. The for loop goes through each character one by one. On each iteration, char holds the current character, which is printed.

4. String Comparison

Strings can be compared using relational (comparison) operators like:

== (equal to)

!= (not equal to)

< (less than)

> (greater than)

<= (less than or equal to)

>= (greater than or equal to)

These comparisons are done using **lexicographical order** (i.e., dictionary order based on Unicode/ASCII values).

Example:

print(“abc” == “abc”)   # True (Both strings are identical, so the result is True.)

print(“abc” != “xyz”)   # True

print(“abc” < “xyz”)   # True(lexicographical order,

compared character by character: ‘a’ < ‘x’ # True → rest of the string isn’t checked.)

print(“abc” > “XYZ”)   # True (Lowercase letters (like ‘a’) have higher Unicode values than uppercase letters (‘X’), so it’s True.)

print(“Apple” < “apple”) # True (uppercase < lowercase, ‘A’ has a lower ASCII/Unicode value than ‘a’, so it returns True.)

5. String Length

Use len() to find the number of characters:

name = “Alice”

print(len(name)) # 5

The string “Alice” contains the characters: ‘A’, ‘l’, ‘i’, ‘c’, ‘e’. Total number of characters = **5.**

So, len(name) returns 5.

6. String Membership Testing

Use in or not in to test if a character or substring is present:

print(“a” in “banana”) # True

print(“z” not in “apple”) # True

2.3.2 String Formatting

In Python, string formatting is used to insert variables or values into strings in a **controlled and readable way**. It allows you to create output that is dynamic and well-formatted, especially useful when displaying data, generating reports, or building user-friendly messages.

Python provides **three main methods** for string formatting:

The % operator (older style)

F-strings (f””) – introduced in Python 3.6

The format() method

2.3.2.1 Using % operator

This method is similar to C-style formatting. Although older, it's still found in legacy code.

Example:

```
name = “Carol”

marks = 88.5

print(“Student: %s, Marks: %.1f” % (name, marks))

# %s = string, %.1f = float with 1 decimal place
```

2.3.2.2 Using f-strings (Python 3.6+)

This is the most modern and concise way introduced in **Python 3.6**. You prefix the string with the letter f and insert variables directly inside {}.

Example:

```
name = “Bob”

score = 90

print(f”{name} scored {score} in the test.”)

# Output: Bob scored 90 in the test.
```

F-strings support **expressions too**:

```
print(f”Next year, {name} will be {age + 1} years old.”)

name = input(“Enter your name: “)

age = int(input(“Enter your age: “))

print(f”Next year, {name} will be {age + 1} years old.”)

#Input:

Enter your name: Asha

Enter your age: 20

# Output:

Next year, Asha will be 21 years old.
```

2.3.2.3 Using str.format() method

This is a widely used and flexible method.

Syntax: “Text {} more text {}”.format(value1, value2)

Example:

```
name = “Alice”

age = 25

print(“My name is {} and I am {} years old.”.format(name, age))

# Output: My name is Alice and I am 25 years old.
```

You can also use **index numbers** inside the placeholders:

```
print(“Name: {0}, Age: {1}, Name again: {0}”.format(name, age))

name = input(“Enter your name: “)

age = input(“Enter your age: “)

print(“Name: {0}, Age: {1}, Name again: {0}”.format(name, age))

#Input

Enter your name: Rahul

Enter your age: 21

#Output

Name: Rahul, Age: 21, Name again: Rahul
```

F-strings are the most preferred due to their readability and efficiency.

2.3.3 String Concatenation

Concatenation is the process of combining two or more strings using the + operator.

Example:

```
first = “Hello”

second = “World”

result = first + “ “ + second

print(result)

# Output: Hello World
```

Note: You cannot concatenate a string and an integer directly.

```
age = 20

print(“Age: “ + str(age))  # Convert integer to string first

# Output: Age: 20
```

2.3.4 String Indexing and Slicing

Strings are one of the most commonly used data types for storing and manipulating text.

A string is essentially a sequence of characters, and Python allows you to access and extract specific parts of a string using techniques called **indexing** and **slicing**. **Indexing** helps retrieve individual characters by their position, while **slicing** allows you to extract substrings using a range of indices. These tools are essential for text processing, such as analyzing words, modifying sentences, or reversing text. Understanding how to use indexing and slicing effectively is a fundamental step in mastering Python programming.

2.3.4.1 Indexing

Each character in a string has an **index**, starting from 0.

Example:

text = “Python”

print(text[0])

# Output: P

Negative Indexing

Python also supports negative indexing. **Negative indexing** starts from -1, which refers to the **last element**, -2 is the second last, and so on.

text = “Python”

print(text[-1])

# Output: n

2.3.4.2 Slicing

Slicing lets you extract parts of a string.

Syntax:

string[start:stop:step]

start – the index to **begin** the slice (inclusive).

stop – the index to **end** the slice (exclusive).

step – the **gap or stride** between elements (default is 1).

All three are **optional**. Omitting them gives flexibility in slicing.

Example:

text = “Python”

print(text[0:3]) # Output: Pyt # extract characters from **index 0 up to, but not including, index 3**.

print(text[:4]) # Output: Pyth # extract characters from **index 0 to 3**

print(text[2:]) # Output: thon #extract characters from **index 2 to the last character**

Recap

- A **string** is a sequence of characters enclosed in single (‘), double (“), or triple (‘’’/”””) quotes.
- **Strings are immutable**—you cannot modify a string in place.
- **Escape sequences** like \n, \t, \”, and \\ help format special characters in strings.
- Use string methods such as:
- lower(), upper(): case conversion
- strip(): trim spaces
- replace(): substitute text
- split() & join(): convert between strings and lists
- find(), index(): locate substrings
- startswith(), endswith(): prefix/suffix check
- Use char in string to **loop through characters** one by one.
- Strings are compared lexicographically using relational operators (==, !=, <, >...).
- Use len() to find the **length** of a string.
- Use in and not in for **membership testing**.
- Format strings using:
- % operator (old)
- f”{}” (modern, preferred)
- .format() method
- Concatenate strings with +, and convert non-strings using str().
- Access individual characters via **indexing**; use **negative indexes** to count from the end.
- Extract substrings using **slicing**: string[start:stop:step].

Objective Type Questions

1. What data type in Python is used to store text?
2. What is the term for the fact that Python strings cannot be modified after creation?
3. Which quotes are used for multi-line strings in Python?
4. Which method converts all characters in a string to uppercase?
5. What method removes leading and trailing whitespace from a string?
6. Which method is used to replace substrings in Python?
7. What method splits a string into a list of words?
8. Which method joins elements of a list into a single string?
9. What method safely finds the index of a substring (returns -1 if not found)?
10. Which method raises an error if a substring is not found?
11. Which function checks if a string starts with a specified substring?
12. What escape sequence is used to insert a new line in a string?
13. What operator is used to compare two strings for equality?
14. What function is used to get the length of a string?
15. What operator is used for string concatenation?
16. What function is used to take user input in Python?
17. What keyword is used in string slicing to define the step?
18. What function converts a value to string for concatenation?
19. What type of loop is commonly used to iterate over a string character by character?
20. What is the most modern and readable way to format strings in Python?

Answers to Objective Type Questions

1. string
2. immutable
3. triple
4. upper

5. strip
6. replace
7. split
8. join
9. find
10. index
11. startswith
12. \n
13. ==
14. len
15. +
16. input
17. step
18. str
19. for
20. f-string

Assignments

1. Write a Python program to input a sentence and count how many times the character ‘e’ appears.
2. Demonstrate the use of at least 5 different string methods on the string text = “ Python is Powerful! “. Explain the purpose of each method used.
3. Check whether a user-input string starts with “Hello” and ends with “!”. Print appropriate messages.
4. Slice the string “Programming” to print the following parts separately: “Program”, “ming”, and “gram”.
5. Write a Python program to accept a string from the user, reverse it using slicing, and check if it is a palindrome.

Reference

1. Lutz, M. (2021). *Learning Python* (5th ed.). O'Reilly Media.
2. Sweigart, A. (2020). *Automate the Boring Stuff with Python* (2nd ed.). No Starch Press.
3. Matthes, E. (2023). *Python Crash Course* (3rd ed.). No Starch Press.
4. Beazley, D., & Jones, B. (2023). *Python Cookbook* (3rd ed.). O'Reilly Media.
5. Downey, A. (2015). *Think Python* (2nd ed.). O'Reilly Media.

Suggested Reading

1. Matthes, E. (2023). *Python crash course: A hands-on, project-based introduction to programming* (3rd ed.).
2. Sweigart, A. (2024). *Automate the boring stuff with Python: Practical programming for total beginners* (3rd ed.).
3. Downey, A. B. (2023). *Think Python: How to think like a computer scientist* (2nd ed.). Green Tea Press.
4. Zelle, J. M. (2022). *Python programming: An introduction to computer science* (3rd ed.). Franklin, Beedle & Associates.
5. Barry, P. (2022). *Head First Python: A brain-friendly guide* (3rd ed.). O'Reilly Media.

Unit 4

List Comprehensions and Iterators

Learning Outcomes

Upon completion of this unit, the learner will be able to:

- define list comprehension and identify its basic syntax.
- recall the syntax used for generator expressions.
- list common iterable objects in Python such as lists, strings, and dictionaries.
- name the two main functions used with iterators: iter() and next().

Prerequisites

Have you ever made a list of things to buy from a shop like apples, bananas, and mangoes? In Python, we can make a similar list using code, and we call it a list. You may also remember using a for loop in Python to go through each item in a list, just like checking off each fruit from your shopping list.

Now, imagine if Python could help you do this in an even shorter and smarter way—like creating a new list of only the items you really need, in just one line of code. That's where list comprehensions and generator expressions come in.

These tools are not only shorter but also make your programs easier to read and understand. List comprehensions help you create new lists from existing ones using a simple and clean format. Generator expressions, on the other hand, allow you to process data efficiently without storing everything in memory especially useful when working with large amounts of data.

Key words

List Comprehension, Generator Expression, Python Loops, Iterator

Discussion

2.4.1 List Comprehension

List comprehension is a simple way to create a new list by taking values from an existing list, changing them if needed, and adding them to the new list, all in a single line of code. It helps make your code shorter, easier to read, and more efficient.

Syntax of List Comprehension:

```
[expression for item in iterable if condition]
```

Where:

- **expression** – What you want to do with each item (e.g., keep it as it is or modify it).
- **item** – The variable that represents each element from the original collection.
- **iterable** – The existing collection (like a list, tuple, or string) you are looping through.
- **condition** (optional) – A test that filters which items to include in the new list.

Consider the following example scenario:

Suppose you have a list of numbers, and you want to create a new list containing only the even numbers from that list.

Without List Comprehension:

We have a list of numbers from 1 to 10. We want to create a new list that contains only the even numbers from this list. The python program for the example scenario is given below:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_numbers = []  # Start with an empty list

for num in numbers:       # Go through each number in the list

    if num % 2 == 0:      # Check if the number is even

        even_numbers.append(num)  # If it is even, add it to the new list

print(even_numbers)
```

Output:

```
[2, 4, 6, 8, 10]
```

The same task can be performed in a simpler and more concise way using list comprehension, as shown in the Python program below:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_numbers = [num for num in numbers if num % 2 == 0]

print(even_numbers)
```

Output:

```
[2, 4, 6, 8, 10]
```

This program uses a for loop to check each number in the list and adds only the even numbers to a new list called even_numbers.

2.4.2 Generator Expression

A generator expression is a short and simple way to create values one by one, like list comprehension. But instead of creating all the values at once and storing them in memory (like a list does), it creates each value only when needed, as you go through it in a loop. This is very helpful when you are working with large amounts of data or when you want to save memory.

Syntax of Generator Expression:

```
    (expression for item in iterable if condition)
```

Generator expressions look similar to list comprehensions, but they use parentheses ( ) instead of square brackets [ ].

Example 1: Square of numbers

Let's say we want to find the squares of numbers from 1 to 5.

```
        squares = (x * x for x in range(1, 6))
```

This line does **not create a list**, but instead creates a **generator object**, which holds the instructions to calculate the square of each number from 1 to 5. To get the values from the generator, we must use a for loop:

```
for num in squares:

  print(num)
```

Output:

```
1

4

9

16

25
```

In this example, the expression x * x is used to calculate the square of each number. The range(1, 6) generates numbers from 1 to 5. The generator expression processes each number one by one, calculates the square, and prints it. This way, the program doesn't create and store the whole list in memory, it only keeps the current value it's working with, making it more memory efficient.

Example 2: Even numbers from 1 to 10

Now let's create a generator to filter even numbers from a list of numbers 1 to 10:

```
numbers = range(1, 11)

evens = (num for num in numbers if num % 2 == 0)

for val in evens:

   print(val)
```

Output:

```
2

4

6

8

10
```

Here, range(1, 11) creates numbers from 1 to 10. The generator expression (num for num in numbers if num % 2 == 0) goes through each number and checks if it is even (i.e., divisible by 2). If the number is even, it is included in the generator. The for loop then prints each even number one at a time. This method is useful for filtering data while keeping the memory usage low.

2.4.2.1 Comparison of List Comprehension and Generator Expressions

Table 2.4.1 Comparison of List Comprehension and Generator Expressions

Features	List Comprehension	Generator Expression
Brackets Used	Uses square brackets [ ]	Uses round brackets ( )
How values are stored	Stores all values at once in memory	Creates one value at a time
Memory usage	Uses more memory	Uses less memory (more efficient)
When to use	When you need all results immediately	When you need values one by one (e.g., in a loop)

2.4.3 Iterators and Iterable Objects

When we write programs in Python, we often deal with collections of data like lists, strings, or dictionaries. Many times, we want to go through each item in these collections one by one. To make this possible, Python provides two important concepts: iterables and iterators. These concepts work behind the scenes whenever you use a for loop. Understanding the difference between them helps you write better and more efficient code, especially when dealing with large amounts of data.

In the following sections, we will explore what iterables and iterators are, how they work, and how they differ from each other, using simple examples.

2.4.3.1 What is an Iterable?

An iterable is any object in Python that contains a collection of items and allows you to go through those items one by one. This means you can access each item in the object one after the other, usually by using a for loop. For example, when you have a list of fruits and you want to print each fruit one by one, you can do that using a for loop because the list is iterable.

Common Examples of Iterables:

- Lists – e.g., [1, 2, 3]
- Tuples – e.g., (1, 2, 3)
- Strings – e.g., “Hello”
- Sets – e.g., {1, 2, 3}
- Dictionaries – e.g., {“name”: “John”, “age”: 25}

Consider the following program:

```
fruits = [“apple”, “banana”, “mango”]

for fruit in fruits:

    print(fruit)
```

In this example, fruits is a list containing three elements: “apple”, “banana”, and “mango”. In Python, lists are iterable objects, meaning they can be traversed one element at a time. By using a for loop, we can go through each item in the list. During each iteration, the variable fruit takes on the value of the current element, and the print() function displays it.

2.4.3.2 What is an Iterator?

In Python, an iterator is a tool that helps us go through items in a group, one at a time. This group can be a list, a string, or a set of items. What makes an iterator special is that it remembers where it left off, so it knows which item to give you next.

To use an iterator, we first need an iterable- an object like a list or string that contains multiple items. We then use the iter() function to turn this iterable into an iterator. Once we have the iterator, we can use the next() function to get the items one after another.

Two Important Functions:

- iter() – Turns a list or other collection into an iterator.

- next() – Gives the next item from the iterator. If there are no more items left, Python shows a StopIteration message.

Example:

```
fruits = [“apple”, “banana”, “mango”]

it = iter(fruits)

print(next(it))

print(next(it))

print(next(it))
```

Output:

```
apple

banana

mango
```

In this example, we first create a list called fruits with three items: “apple”, “banana”, and “mango”. We then use the iter() function to turn the list into an iterator and store it in the variable it. Using the next() function, we get each item one by one from the iterator. The first call to next(it) gives “apple”, the second gives “banana”, and the third gives “mango”. If we try to call next(it) again after this, Python shows a StopIteration error because there are no more items left in the list.

Recap

- In Python, a list is a common example of an iterable object, which means it can be looped over using a for loop.
- The for loop is used to access each element in a list or any iterable, allowing sequential processing of items.
- List comprehensions offer a concise way to create new lists by combining loops and conditions into a single line of code.
- A list comprehension typically includes an expression, a loop, and an optional condition to filter elements.
- For example, [x for x in numbers if x % 2 == 0] generates a list of even numbers from an existing list.
- Generator expressions are similar to list comprehensions but use parentheses instead of square brackets and generate items one at a time (lazily).
- The built-in function iter() is used to create an iterator from an iterable, and next() retrieves the next item from the iterator.

Objective Type Questions

1. What is the output type of a list comprehension?
2. What do generator expressions use instead of square brackets?
3. Which function is used to get the next item from an iterator?
4. What exception is raised when an iterator has no more items?
5. What function is used to convert an iterable into an iterator?
6. What type of object does a generator expression return?
7. Are strings iterable in Python?
8. What type of comprehension uses [ ]?
9. Can generator expressions be used in a for loop directly?
10. What Python data structure allows filtering and looping in one line?

Answers to Objective Type Questions

1. List
2. Parentheses()
3. next
4. StopIteration
5. iter
6. Generator
7. Yes
8. List
9. Yes
10. Comprehension

Assignments

1. Explain with examples how list comprehensions work in Python.
2. Write a Python program to generate a list of even numbers from 1 to 20 using list comprehension.
3. Differentiate between iterators and iterable objects in Python with suitable examples.
4. What is a generator expression? How is it different from a list comprehension?
5. Using the iter() and next() functions, write a program to manually access each element in a list of five fruits.

Reference

1. https://www.w3schools.com/python/
2. https://www.learnpython.org/

Suggested Reading

1. Brown, Martin C. Python: The complete reference. Osborne/McGraw-Hill, 2001.
2. Jose, Jeeva. Taming Python by Programming. KHANNA PUBLISHING HOUSE.
3. Lutz, Mark. Learning python: Powerful object-oriented programming. " O'Reilly Media, Inc.", 2013.

3

BLOCK

Functions, Modules, Packages and Regular Expressions

Unit 1
Functions

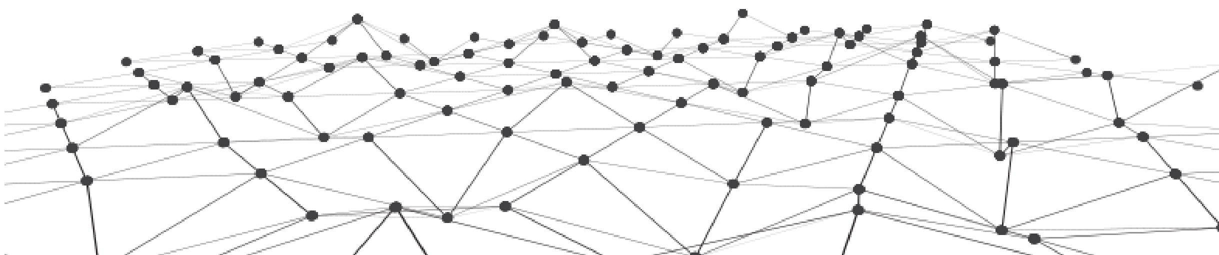
Learning Outcomes

After completing this section, learners will be able to:

- familiarize the concept of functions and explain their role in structuring and organizing Python programs efficiently.
- differentiate between built-in functions and user-defined functions, and apply them appropriately in various coding scenarios.
- demonstrate the ability to define functions using different types of arguments such as positional, keyword, default, variable-length, and unpacked arguments.
- analyze how Python handles function arguments with respect to mutable and immutable data types using the concepts of pass by reference and pass by value.
- develop and use advanced function techniques such as recursive functions, lambda (anonymous) functions, and nested functions to solve real-world programming problems.

Prerequisites

Consider a program that needs to calculate the area of multiple rectangles at different points in the code. Writing the same formula repeatedly not only clutters the program but also increases the chance of errors. As Python programs grow larger and more complex, writing all the code in a single block becomes difficult to manage. Repeating the same logic in multiple places leads to redundancy and makes maintenance harder. Functions help break down a large problem into smaller, manageable tasks, making the code more structured. They allow code to be reused by defining it once and calling it multiple times with different inputs. This improves readability, reduces errors, and simplifies debugging. Functions also support better testing and make collaboration easier by organizing code into independent, logical units. Overall, functions promote modularity, efficiency, and clarity in Python programming.



Key words

Def, Return, Arguments, Parameters, Scope, Lambda, Recursion, Built-in Functions

Discussion

A function is a block of statements that performs a particular task. The main idea behind the function is to put some commonly or repeatedly done tasks together and can use functions whenever we need to perform the same task multiple times without writing the same code again. As our program grows larger and larger, functions make it more organized and manageable.

3.1.1 Advantages of functions

Modularity: Functions break down complex programs into smaller, manageable parts, making code more understandable, testable, and maintainable. They promote code reusability and the principle of avoiding repetition.

Code Organization: Functions provide a structured approach to programming by dividing code into logical units. This improves overall code navigation, comprehension, and reduces the likelihood of errors. Well-organized code is easier to read and maintain.

Reusability: Functions enable code reuse, allowing them to be called multiple times with different inputs. This eliminates the need for redundant code, promoting efficient programming practices.

Abstraction: Functions hide internal implementation details, allowing users to focus on what the function does and how to use it. This simplifies programming and promotes a higher-level understanding of the program's functionality.

Testing and Debugging: Functions facilitate testing and debugging efforts as they can be individually tested to ensure proper functionality. Debugging becomes more manageable as errors can be isolated to specific functions, enabling focused troubleshooting.

Collaboration: Functions promote collaboration by dividing code into independent units, enabling different team members to work on separate functions concurrently. Functions also facilitate code sharing and open-source collaboration within the Python community.

Code Maintainability: Functions enhance code maintainability by isolating specific tasks. Updates or changes can be made to individual functions, reducing the risk of introducing bugs. This simplifies the maintenance process, particularly in large and complex projects.

3.1.2 Types of function

There are two types of functions:

- **Built-in functions**
- **User defined functions**

3.1.2.1 Built-in Functions

Built-in functions are an integral part of Python's standard library, offering a broad range of functionalities that streamline coding tasks. These functions are readily accessible without the need for additional installation or setup. An example of such a fundamental built-in function is print(), which enables us to display text or variables on the console. By simply passing the desired content as arguments, we can swiftly output information to the user. Executing print("Hello, world!") will print the phrase "Hello, world!" on the console.

Python's built-in functions provide numerous ways to manipulate and analyze data. One such function is sorted(), which returns a new list containing the sorted elements from the input iterable. Using sorted([5, 2, 7, 1, 3]) will yield [1, 2, 3, 5, 7], showcasing how the function arranges the elements in ascending order. Another useful built-in function is len(), which determines the length of an object, such as a string, list, or tuple. By employing len(), we can quickly ascertain the number of elements in a given collection. For example, len("Python") will return the value 6, representing the length of the string "Python".

Python's built-in functions also facilitate convenient mathematical operations. The abs() function, for instance, returns the absolute value of a number, disregarding its sign and providing the positive value. Hence, executing abs(-5) will yield 5. Additionally, the round() function allows us to round a floating-point number to a specified number of decimal places. For example, round(3.14159, 2) will return 3.14, rounding the number to two decimal places. These built-in mathematical functions offer flexibility when performing calculations and are commonly employed in various applications.

3.1.2.2 User Defined functions

User-defined functions are those functions that we define ourselves to do certain specific tasks. A user-defined function in Python is a piece of code that you create to perform a particular task. It enables you to group a set of instructions together under a unique name, enhancing the modularity and organization of your code. This named entity becomes a custom function within your program, allowing you to call it repeatedly with different inputs.

3.1.3 Creating a Function

We can define our own functions in Python by using the "def" keyword.

The syntax of a Python function is

```
def function_name(parameter1, parameter2, ...):
"""
 Docstring: Description of the function (optional).
"""
```

# Function body : Code block that defines the behavior of the function

# This can include variable declarations, conditional statements, loops, etc.

# Optional return statement to specify the value(s) to be returned called return value

To define a function in Python, we use the **“def”** keyword, followed by the **function name,** which should be a valid identifier in Python. If the function takes any parameters, they are placed within parentheses and separated by commas.

A docstring, which is an optional multi-line string enclosed in triple quotes (“””), can be included right after the function definition. This docstring provides a brief description of the function’s purpose, parameters, and return values.

The function body comprises the code block that specifies the behavior of the function. It starts with a colon (:) and is indented consistently. The body can contain multiple statements, all indented at the same level.

If the function is expected to return a value, the “return” statement is used to specify the value(s) to be returned. The return statement is optional, and if it is not included, the function will automatically return None. It is possible to return a single value or multiple values separated by commas.

Here is a simple python function definition:

```
def greet(name):  """ Prints a greeting message."""

print("Hai, " + name + "!")
```

In this example, we have defined a function called greet that takes a parameter called name. The function’s purpose is to print a greeting message to the console. When the function is called with a specific name, it will print “Hai, “ followed by the provided name and an exclamation mark.

3.1.4 Calling a Function

To call a function in Python, you simply write the function name followed by parentheses. If the function requires any arguments, you provide them inside the parentheses.

To call the above example function, we write:

greet(“Ann”)

Here the passed string argument is “Ann”. The function is then executed, resulting in the output “Hai, Aan!” being displayed on the console.

3.1.5 Arguments

Arguments are the values that you pass to a function during its invocation. They serve as inputs for the function to perform specific operations or calculations. Python supports various types of arguments which are included below.

3.1.5.1 Positional Arguments

These arguments are provided to a function in the same order as they are defined in the function's parameter list. The values are assigned to the respective parameters based on their positions.

Example:

```
def add_numbers(x, y):
 """Adds two numbers."""
        return x + y
result = add_numbers(3, 5)
print(result)
# Output: 8
```

In this example, 3 is assigned to x and 5 is assigned to y based on their positions.

3.1.5.2 Keyword Arguments

With keyword arguments, you explicitly specify the parametername followed by the corresponding value, separated by an equal sign. This allows you to pass arguments in any order, disregarding their position in the parameter list.

Example:

```
def add_numbers(x, y):
 “””Adds two numbers.”””
 return x + y
result = add_numbers(y=4, x=2)
print(result)
# Output: 6
```

Here, the function is called with keyword arguments, allowing us to specify the values explicitly.

3.1.5.3 Default Arguments

Default arguments have predefined values assigned to them in the function's parameter list. If an argument is not supplied during the function call, the default value is used instead.

Example:

```
def greet(name, message=”Hai”):
 “””Prints a personalized greeting.”””
 print(message + “, “ + name)
greet(“Ann”) # Output: Hai, Ann
greet(“Balu”, “Hello”)
# Output: Hello, Balu
```

In this example, the message parameter has a default value of “Hai”. If not provided, the default value is used.

3.1.5.4 Variable-length Arguments

Python functions can accept a varying number of arguments. To achieve this, you can use the asterisk (*) before a parameter name for variable-length positional arguments, or two asterisks (**) for variable-length keyword arguments.

Example:

```
def add(*numbers):
 “””Addition of numbers.”””
 total = sum(numbers)
 return total
result = add(1, 2, 3, 4, 5)
print(result)
# Output: 15
```

The function add accepts a variable number of positional arguments using *numbers. The arguments are treated as a tuple inside the function.

3.1.5.5 Unpacking Arguments

Arguments can be unpacked from a list or tuple using the asterisk (*) operator. This allows you to pass the individual elements of a sequence as separate arguments to a function.

Example:

```
def add(a, b, c):
 “””Adds three numbers.”””
 return a + b + c
numbers = [2, 3, 4]
result = add(*numbers)
print(result)
```

```
# Output: 9
```

In this example, the elements of the numbers list are unpacked using * and passed as separate arguments to the add function.

3.1.6 Pass by Reference and Pass by Value

In languages such as C++ or Java, it's important to know whether function arguments are passed by value or by reference. Python, however, uses a different approach that can be confusing. It doesn't strictly use either method. Instead, Python employs a model known as **pass by object reference**, or sometimes **call by sharing**.

3.1.6.1 Pass by Reference (Mutable Objects):

In Python, when an object is passed as an argument to a function using pass by reference, a reference to the object's memory location is passed. This means that any modifications made to the object within the function will impact the original object outside the function as well.

However, it's important to note that in Python, all variable assignments are references to objects. So when an object is passed to a function, a reference to the object is passed as well. If the function modifies the object directly, the changes will be visible outside the function since it operates on the same underlying object.

Example:

```
def modify_list(lst):

 lst.append(4) # Modifying the list within the function

test_list = [1, 2, 3]

modify_list(test_list)

print(test_list)

# Output: [1, 2, 3, 4]
```

In this example, the test_list object (a list) is passed to the modify_list function. The function modifies the list by appending an element. Since lists are mutable objects, the changes made to the list within the function are also reflected in the original list outside the function.

3.1.6.2 Pass by Value (Immutable Objects):

When using pass by value, a copy of the object's value is passed as an argument to a function. This means that any modifications made to the object within the function do not affect the original object outside the function.

However, it's important to note that in Python, objects of immutable types, such as integers, strings, and tuples, are passed by value. If the function modifies the object directly, a new object is created, while the original object remains unchanged.

Example:

```
def modify_number(num):
 num += 1 # Modifying the number within the function
test_number = 5
modify_number(test_number)
print(test_number)
# Output: 5
```

In this example, the test_number object (an integer) is passed to the modify_number function. However, integers are immutable objects in Python. Therefore, any modifications made to the num variable within the function do not affect the original test_number object outside the function.

3.1.7 Scope and Lifetime of Variables

Scope and lifetime of variables determine where and for how long a variable is accessible and exists in a program.

3.1.7.1 Scope

Global Scope: Variables defined outside any function or class have global scope, meaning they can be accessed from anywhere within the program.

Local Scope: Variables defined inside a function or block have local scope, which means they are only accessible within that specific function or block.

Lifetime

Global Variables: Global variables are created when the program starts and persist throughout the entire execution of the program. They are destroyed when the program terminates.

Local Variables: Local variables have a limited lifetime within the scope of the function or block in which they are defined. They are created when the function or block is entered and cease to exist when the function or block is exited.

Example:

```
def test_function():
local_var = 15 # Local variable within the function
print("Local variable:", local_var)
global_var = 25 # Global variable
test_function()
print("Global variable:", global_var)
```

Output

```
Local variable: 15

Global variable: 25
```

In this example, we have a function called test_function() that defines a local variable local_var with a value of 15. This variable is only accessible within the scope of the function. When the function is called, the local variable is created and printed.

We also have a global variable global_var defined outside the function. Global variables are accessible from anywhere in the program. It is printed after the function call.

When we run the program, the output will be:

```
Local variable: 15

Global variable: 25
```

Here, we can see that the local variable local_var is accessible and exists within the scope of the function test_function(). Once the function finishes executing, the local variable is destroyed.

On the other hand, the global variable global_var has a global scope, and it persists throughout the entire program execution.

3.1.8 Return Values

Return values in Python pertain to the values that a function can provide to the caller once it has executed its tasks. The return statement is employed to indicate the specific value that a function will return.

3.1.8.1 Single Value Return:

To return a single value, a function employs the return statement followed by the value that will be returned. This allows the function to provide a single result to the caller.

Example:

```
def multiply(a, b):

return a * b

result = multiply(3, 4)

print(result)

# Output: 12
```

By utilizing the return statement, the multiply function returns the product of two numbers, and the resulting value is assigned to the variable "result."

Multiple Value Return: Functions have the ability to return multiple values by listing them separated by commas within the return statement. This allows the function to provide multiple results as a tuple or any other sequence type.

Example:

```
def get_person_details():
name = “Ann”
age = 25
occupation = “Teacher”
return name, age, occupation
person = get_person_details()
print(person)
# Output: (“Ann”, 25, “Teacher”)
```

3.1.8.2 Empty Return

In situations where a return statement is encountered without a value or if a function lacks a return statement, Python implicitly returns None.

Example:

```
def is_even(number):
 if number % 2 == 0:
 return True
          else:
          return
result1 = is_even(4)
result2 = is_even(5)
print(result1) # Output: True
print(result2) # Output: None
```

In this example, we have a function called is_even() that checks whether a given number is even. If the number is divisible by 2, the function returns True using the return True statement. If the number is not even, the function does not explicitly provide a return value.

When we call is_even() with the number 4, the function returns True, indicating that 4 is an even number. We assign the return value to the variable result1 and print it, which outputs True.

When we call is_even() with the number 5, which is an odd number, the function does not have a return statement for this case. In such situations, Python implicitly returns

None. We assign the return value to the variable result2 and print it, which outputs None.

3.1.9 Function within Functions

In Python, it is permissible to define a function within another function, which is referred to as a nested function or a function within a function. This allows for the creation of a local function that can only be accessed and invoked from within the enclosing function. The inner function has visibility and access to the variables and parameters of the outer function, forming a nested scope.

Example:

```
def outer_function():
 def inner_function():
 print("This is the inner function")
 print("This is the outer function")
          inner_function()
outer_function()
```

In this example, the outer_function defines the inner_function within it. When the outer_function is called, it prints "This is the outer function" and then invokes the inner_function. The inner_function, in turn, prints "This is the inner function".

3.1.10 Anonymous Functions

Anonymous functions in Python are also known as lambda functions. They are compact and inline functions that can be defined without the traditional "def" keyword. Instead, lambda functions are created using the "lambda" keyword, followed by a parameter list, a colon (:), and an expression that defines the function's behavior.

The basic syntax of a lambda function can be summarized as follows:

3.1.11 lambda arguments: expression

Lambda functions are commonly used for simple, one-line operations, especially in situations where a full function definition is not necessary or practical. They are often employed as arguments to other functions or utilized within functional programming paradigms.

For example, consider a lambda function that calculates the square of a given number:

```
square = lambda x: x ** 2
result = square(5)
print(result)
# Output: 25
```

In this example, we define a lambda function named "square" that takes an argument "x" and returns its square. We then call the lambda function with the argument "5" and store the result in the variable "result". Finally, we print the value of "result", which outputs "25".

Recursive Function

A recursive function is a function that invokes itself during its execution. It is employed when a problem can be divided into smaller, similar subproblems. With each recursive call, the function addresses a reduced version of the problem until a base case is encountered, which serves as the stopping condition for the recursion.

Example:

```
def factorial(n):
 if n == 0:
 return 1
 else:
        return n * factorial(n - 1)
```

In this example, the factorial function takes an integer n as an argument. It checks if n is equal to 0, which represents the base case. If it is, the function returns 1. Otherwise, it recursively calls itself with the argument n - 1 and multiplies the result by n. This process continues until the base case is reached.

Let's use this function to calculate the factorial of 5:

```
result = factorial(5)
print(result)
# Output: 120
```

In this case, we call the factorial function with the argument 5 and assign the result to the variable result. The factorial of 5 is calculated by recursively multiplying 5 by the factorial of 4, which further multiplies 4 by the factorial of 3, and so on, until we reach the base case. The final result, 120, is then printed.

Recap

- A function is a block of reusable code that performs a specific task, helping avoid code repetition.
- Functions divide large programs into smaller, manageable parts, making the code easier to understand and test.
- Functions help structure code into logical sections, improving readability and maintainability.
- Once defined, a function can be called multiple times with different inputs, reducing redundancy.

- Functions hide the implementation details and allow the user to focus on what the function does.
- Functions can be tested individually, making it easier to isolate and fix errors.
- Code can be divided into functions so multiple team members can work on different tasks concurrently.
- Python supports built-in functions (like print(), len(), abs()) and user-defined functions created using the def keyword.
- Functions are defined using def, followed by the function name, parentheses (with parameters), and a colon.
- To execute a function, call it using its name followed by parentheses, optionally passing arguments.
- Arguments are inputs passed to functions. Types include:
  - Positional (ordered)
  - Keyword (named)
  - Default (with default values)
  - Variable-length (*args, **kwargs)
  - Unpacking (using * to unpack sequences)
- Mutable types (like lists) are passed by reference, and changes inside the function affect the original object.
- Immutable types (like strings and integers) behave as if passed by value; changes inside the function do not affect the original object.
- Variables defined outside functions have global scope and can be accessed throughout the program.
- Variables defined inside functions have local scope and only exist during the function's execution.
- The return statement is used to send data back from a function. Functions can return a single value, multiple values, or None.
- Functions can be defined inside other functions. These are called nested or inner functions.
- Lambda (anonymous) functions are short, one-line functions created with the lambda keyword.

- Example of lambda function:

  square = lambda x: x ** 2

  print(square(5)) # Output: 25

- Recursive functions call themselves to solve problems that can be broken into smaller similar problems (e.g., factorial).

Objective Type Questions

1. What keyword is used to define a function in Python?
2. What type of function is print() in Python?
3. What is the term for functions defined by the user?
4. Which keyword is used to return a value from a function?
5. What type of argument allows passing values without considering order?
6. What symbol is used for variable-length positional arguments?
7. What kind of function is defined without a name using the lambda keyword?
8. What is the default return value of a function that has no return statement?
9. What type of scope does a variable defined inside a function have?
10. What is the process of calling a function from within itself called?
11. Which function returns the length of a string or list?
12. Which function returns the absolute value of a number?
13. What is the keyword used to define an anonymous function?
14. What is the term used when a function is defined inside another function?
15. What is the name of the model used in Python for argument passing?

Answers to Objective Type Questions

1. def
2. Built-in

3. User-defined
4. return
5. Keyword
6. *
7. Lambda
8. None
9. Local
10. Recursion
11. len
12. abs
13. lambda
14. Nested
15. Sharing

Assignments

1. Explain the advantages of using functions in Python programming. Discuss how modularity, reusability, code organization, abstraction, and maintainability contribute to better software design. Provide examples to support your explanation.

2. Differentiate between built-in functions and user-defined functions in Python. Define each type, give at least three examples for built-in functions, and explain the syntax and use of user-defined functions with suitable code.

3. Discuss various types of function arguments supported in Python. Elaborate on positional, keyword, default, variable-length, and unpacked arguments. Include sample programs to demonstrate how each type works.

4. What is the difference between pass by value and pass by reference in Python? With the help of appropriate examples, explain how mutable and immutable objects behave when passed to a function in Python.

5. Explain scope and lifetime of variables in Python with examples. Describe global and local scope and variable lifetimes using code snippets. Also, explain how scope affects variable access inside and outside functions.

6. Define and compare recursive functions, lambda functions, and nested functions. Explain the concept of each, including syntax and use-cases. Provide examples to show how and when each type of function can be used effectively.

Unit 2

Built-in Functions and Lambda Functions

Learning Outcomes

After completing this section, learners will be able to:

- explore Python's built-in numeric and mathematical functions.
- explain the use of string methods for text manipulation.
- utilize list methods for data operations.
- perform type conversions in Python.
- familiarize lambda functions.

Prerequisites

Have you ever wondered how Python can instantly tell you the maximum of a list of numbers, or change a string to uppercase with just a single word? Python comes with a powerful set of built-in tools that make coding faster, cleaner, and more enjoyable. These features are used every day by developers to solve real-world problems with minimal effort.

As you explore Python further, you'll begin to notice how often you repeat certain tasks like cleaning up text, organizing data in a list, or transforming values. Wouldn't it be great if Python had shortcuts for these common actions? The good news is, it does! And learning them not only saves time but also makes your code look more professional.

There's also something special about writing functions without even giving them a name - yes, that's possible! With just a few keystrokes, you can embed logic directly where it's needed. These compact expressions can change the way you think about programming. Ready to unlock Python's hidden potential? Let's begin.

Key words

abs( ), round( ), pow( ), upper( ), lower( ), append( ), remove( ), Lambda functions

Discussion

3.2.1 Python Built-in Functions

Python simplifies mathematical operations by providing a rich set of built-in functions that can be used directly, without importing any external modules. These functions handle common numeric tasks such as calculating absolute values, powers, minimum and maximum values, rounding, and more. Whether you're working with integers, floating-point numbers, or performing basic arithmetic operations, Python's built-in numeric and mathematical functions offer reliable and efficient tools to support your programming needs. Understanding these functions is essential for writing clean, concise, and effective code.

3.2.1.1 Numeric and Mathematical Functions

Python provides several built-in numeric and mathematical functions that allow you to perform common mathematical operations without importing any external modules. These functions are simple yet powerful tools for performing calculations, manipulating numbers, and analyzing data.

1. abs( ) – Absolute Value

Returns the absolute (non-negative) value of a number.

abs(-7) → 7

abs(3.5) → 3.5

2. round( ) – Rounding Numbers

Rounds a floating-point number to the nearest integer or to a specified number of decimal places.

round(3.14159) → 3

round(3.14159, 2) → 3.14

3. pow( ) – Exponentiation

Returns the value of a number raised to the power of another number.

pow(2, 3) → 8 # Same as 2^3

pow(2, 3, 5) → 3 # (2^3) % 5 = 8 % 5 = 3

4. min ( ) – Minimum Value

Returns the smallest value among multiple values or in an iterable.

min(5, 3, 9) → 3

min([4, 1, 7]) → 1

5. max( ) – Maximum Value

Returns the largest value among multiple values or in an iterable.

max(5, 3, 9) → 9

max([4, 1, 7]) → 7

6. sum( ) – Sum of Elements

Returns the total of all numeric values in an iterable (like a list or tuple).

sum([1, 2, 3, 4]) → 10

sum((5, 5, 5)) → 15

7. divmod( ) – Quotient and Remainder

Returns a tuple containing the quotient and the remainder when dividing two numbers.

divmod(10, 3) → (3, 1)   # 10 // 3 = 3, 10 % 3 = 1

8. bin( ), oct( ), hex( ) – Number Base Conversions

bin( ) – Converts an integer to binary format

oct( ) – Converts an integer to octal format

hex( ) – Converts an integer to hexadecimal format

bin(10) → '0b1010'

oct(10) → '0o12'

hex(10) → '0xa'

3.2.1.2 Built-in String methods in Python

Strings in Python come with a variety of built-in methods that allow easy manipulation, searching, and formatting. These methods return new strings or values without modifying the original string (since strings are immutable).

1. upper() – Convert to Uppercase

Converts all characters in the string to uppercase.

"hello".upper() → "HELLO"

2. lower() – Convert to Lowercase

Converts all characters in the string to lowercase.

"HELLO".lower() → "hello"

3. capitalize() – Capitalize first letter

Capitalizes the first character and makes the rest lowercase.

"python".capitalize() → "Python"

4. title() – Capitalize each word

Capitalizes the first letter of each word in the string.

“hello world”.title() → “Hello World”

5. strip() – Remove Whitespace

Removes leading and trailing whitespace.

“ hello “.strip() → “hello”

6. replace(old, new) – Replace Substring

Replaces all occurrences of a substring with another.

“banana”.replace(“a”, “o”) → “bonono”

7. find(sub) – Find Substring

Returns the index of the first occurrence of the substring. Returns -1 if not found.

“hello”.find(“e”) → 1

8. count(sub) – Count Substring

Counts how many times a substring occurs in the string.

“banana”.count(“a”) → 3

9. startswith(prefix) / endswith(suffix)

Checks whether the string starts or ends with a particular substring.

“python”.startswith(“py”) → True

“notes.txt”.endswith(“.txt”) → True

10. split(sep) – Split into List

Splits the string into a list based on a separator (default is space).

“one,two,three”.split(“,”) → [‘one’, ‘two’, ‘three’]

11. join(iterable) – Join Elements with Separator

Joins elements of an iterable into a single string with the current string as a separator.

“-”.join([“a”, “b”, “c”]) → “a-b-c”

12. isalpha(), isdigit(), isalnum() – Character Checks

isalpha() – True if all characters are alphabetic.

isdigit() – True if all characters are digits.

isalnum() – True if all characters are alphanumeric.

“abc”.isalpha() → True

```
“123”.isdigit() → True
“abc123”.isalnum() → True
```

3.2.1.3 Built-in List methods

1. append(x) – Add item to end

Adds an item x to the end of the list.

```
list1 = [1, 2, 3]
list1.append(4)
print(list1) → [1, 2, 3, 4]
```

2. extend(iterable) – Add multiple items

Adds all elements of an iterable (like a list or tuple) to the end.

```
list1 = [1, 2, 3]
list1.extend([4, 5])
print(list1) → [1, 2, 3, 4, 5]
```

3. insert(i, x) – Insert at position

Inserts item x at position i.

```
list1 = [1, 2, 4]
list1.insert(2, 3)
print(list1) → [1, 2, 3, 4]
```

4. remove(x) – Remove item by value

Removes the first occurrence of item x.

```
list1 = [1, 2, 3, 2]
list1.remove(2)
print(list1) → [1, 3, 2]
```

5. pop([i]) – Remove and return item

Removes and returns the item at index i. If i is not given, removes the last item.

```
list1 = [1, 2, 3]
list1.pop() → 3
print(list1) → [1, 2]
list1.pop(0) → 1
```

```
print(list1) → [2]
```

6. index(x) – Find Index of Item

Returns the index of the first occurrence of item x.

```
list1 = [10, 20, 30]

list1.index(20) → 1
```

7. count(x) – Count Occurrences

Returns the number of times item x appears in the list.

```
list1 = [1, 2, 2, 3, 2]

list1.count(2) → 3
```

8. sort() – Sort List

Sorts the list in ascending order (modifies the list in-place).

```
list1 = [3, 1, 2]

list1.sort()

print(list1) → [1, 2, 3]
```

9. reverse() – Reverse the List

Reverses the order of elements in-place.

```
list1 = [1, 2, 3]

list1.reverse()

print(list1) → [3, 2, 1]
```

10. clear() – Remove All Items

Removes all elements from the list.

```
list1 = [1, 2, 3]

list1.clear()

print(list1) → []
```

3.2.1.4 Type Conversion

Type conversion functions allow you to convert data from one type to another. These are especially useful when handling user input, file data, or performing operations between different data types.

int() – Converts a value to an integer.

Example: int('5') → 5

float() – Converts a value to a floating-point number.

Example: float('3.14') → 3.14

str() – Converts a value to a string.

Example: str(100) → '100'

list(), tuple(), dict(), set() – Convert iterables to their respective data structure types.

Example: list('abc') → ['a', 'b', 'c']

3.2.2 Lambda Function

A Python lambda function is an anonymous, compact function created using the lambda keyword. Lambda functions are useful in situations where you need a short, throwaway function for a limited purpose.

3.2.2.1 Why is the Lambda function needed?

1. Concise Function Definition : Lambda functions allow you to define small functions in a single line, which makes your code more concise and readable

2. Ideal for One-Time Use : Lambda functions are often used when a function is needed temporarily and does not require a formal name.

3. Useful in Functional Programming : Lambda functions work well with functions like map( ), filter( ), and reduce( ), which expect another function as an argument.

4. Simplifies Callbacks and Event Handlers : In GUI applications or asynchronous programming, lambda helps define quick callback functions.

5. Cleaner Code for Simple Operations: Avoids the overhead of formally defining a function when a single expression will suffice.

Unlike regular functions defined with def, lambda functions have no name and consist of a single expression whose result is automatically returned.

Syntax

```
lambda arguments : expression
```

Where

arguments are the input parameters to the lambda function. A lambda function can take any number of arguments, separated by commas and

expression is the single expression that the lambda function evaluates and returns.

Example

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

Lambda functions are not a replacement for regular functions but are needed for brevity, clarity, and convenience in specific situations where a short, unnamed function is sufficient.

3.2.2.2 Use cases of Lambda function

1. Used with Built-in Functions like map(), filter(), and reduce()

These functions expect another function as an argument. Lambda functions are often used here because the logic is simple and doesn't require a named function.

map( ) – Applies a function to every element in a list (or any iterable).

```
nums = [1, 2, 3, 4]

squares = list(map(lambda x: x * x, nums))

print(squares)    # Output: [1, 4, 9, 16]
```

☐ Here, lambda x: x*x computes the square of each number in the list.

filter( ) – Filters elements based on a condition.

```
nums = [1, 2, 3, 4, 5]

evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens)  # Output: [2, 4]
```

☐ This lambda returns True for even numbers, so only those are included.

reduce( ) – Repeatedly applies a function to the items of a list to reduce it to a single

value. (You must import it from functools)

```
from functools import reduce

nums = [1, 2, 3, 4]

product = reduce(lambda x, y: x * y, nums)

print(product)  # Output: 24
```

☐ This multiplies all elements together using the lambda.

2. Sorting with custom keys

Lambda is often used in the key argument of sorted( ) when sorting objects based on a certain rule.

```
names = ['john', 'Alice', 'bob']

sorted_names = sorted(names, key=lambda x: x.lower())

print(sorted_names)  # Output: ['Alice', 'bob', 'john']
```

☐ Here, sorting is done alphabetically without case sensitivity using lambda x: x.lower().

3. Short functions for GUI or Callback-based Code

In GUI or event-driven programs, lambda can define quick actions (callbacks) without a separate function.

```
button_click = lambda: print("Button clicked")

button_click()  # Output: Button clicked
```

☐ A lambda is used to define what should happen when a button is clicked—no need for a separate named function.

4. Inline function in List Comprehensions

You can use lambda inside comprehensions when applying simple operations to each element.

```
add_five = lambda x: x + 5

updated = [add_five(i) for i in range(3)]

print(updated)  # Output: [5, 6, 7]
```

☐ lambda x: x + 5 quickly adds 5 to each value without a full function definition.

Recap

Numeric and Mathematical Functions

- abs(x) – Absolute value
- round(x[, n]) – Round to nearest integer or given decimal places
- pow(x, y[, z]) – Exponentiation, optional modulus
- min() / max() – Smallest or largest among values or in iterable
- sum(iterable) – Sum of all values
- divmod(a, b) – Tuple of quotient and remainder
- bin(x) / oct(x) / hex(x) – Convert integer to binary, octal, hexadecimal

Built-in String Methods

- upper() / lower() – Convert to uppercase or lowercase
- capitalize() / title() – Capitalize first letter or each word
- strip() – Remove leading/trailing whitespace

- replace(old, new) – Replace substring
- find(sub) – Index of first occurrence, -1 if not found
- count(sub) – Number of occurrences of a substring
- startswith() / endswith() – Check prefix or suffix
- split(sep) – Split into list using separator
- join(iterable) – Join elements with separator
- isalpha() / isdigit() / isalnum() – Character checks

Built-in List Methods

- append(x) – Add item to end of list
- extend(iterable) – Add multiple elements
- insert(i, x) – Insert item at index i
- remove(x) – Remove first occurrence
- pop([i]) – Remove and return item (default last)
- index(x) – Return index of first occurrence
- count(x) – Count occurrences of value
- sort() – Sort list in ascending order
- reverse() – Reverse list in-place
- clear() – Remove all elements from list

Type Conversion Functions

- int() / float() / str() – Convert to integer, float, string
- list() / tuple() / dict() / set() – Convert to respective collection types

Lambda Function

- Anonymous, one-line function
- Defined using lambda keyword
- Syntax: lambda arguments: expression
- Returns result of expression automatically
- Ideal for short, throwaway logic

- Useful in functional and event-driven programming

Why Lambda is Needed

- Concise function definition in a single line
- Good for temporary use
- Clean integration with map(), filter(), reduce()
- Simplifies GUI callbacks and event handlers
- Reduces code clutter for simple logic

Use Cases of Lambda Function

With map(), filter(), reduce()

- map(lambda x: x*x, iterable) – Apply operation to each element
- filter(lambda x: condition, iterable) – Keep elements meeting condition
- reduce(lambda x, y: x*y, iterable) – Combine all items into one value

With sorted() and custom keys

- sorted(list, key=lambda x: x.lower()) – Case-insensitive sorting

GUI callbacks or inline actions

- lambda: action() – Quick inline function for events

In list comprehensions

- lambda x: x + 5 – Short arithmetic operation on elements

Objective Type Questions

1. Which function returns the absolute value of a number?
2. Which keyword is used to define a lambda function?
3. Which string method converts all characters to lowercase?
4. Which list method removes the last element by default?
5. Which function returns both quotient and remainder?
6. Which function converts an integer to binary?

7. Which type conversion function converts a value to a float?
8. Which list method is used to add multiple elements at once?
9. Which function is used to get the sum of a list?
10. Which string method checks if all characters are digits?
11. Which function returns the highest value among inputs?
12. Which string method returns the number of occurrences of a substring?
13. Which list method is used to sort elements?
14. Which string method splits a string into a list?
15. Which function is used to round a number?
16. Which string method replaces a specific substring?
17. Which function performs exponentiation?
18. Which list method reverses the order of elements?
19. Which string method capitalizes the first character of each word?
20. Which function is commonly used with lambda for applying logic to every item in a list?

Answers to Objective Type Questions

1. abs
2. lambda
3. lower
4. pop
5. divmod
6. bin
7. float
8. extend

9. sum
10. isdigit
11. max
12. count
13. sort
14. split
15. round
16. replace
17. pow
18. reverse
19. title
20. map

Assignments

1. Discuss Python's built-in numeric and mathematical functions. Explain the purpose and usage of functions like abs( ), round( ), pow( ), min( ), max( ), and sum( ) with appropriate examples.

2. Describe the common built-in string methods available in Python. How do methods like upper( ), lower( ), strip( ), replace( ), find( ), and split( ) help in string manipulation? Illustrate with examples.

3. Explain the different list methods in Python with examples. How do methods like append( ), extend( ), insert( ), remove( ), pop( ), and sort( ) support list operations?

4. What is a lambda function in Python? Why is it needed? Write the syntax and explain its use with at least three different use cases, such as with map( ), filter( ), and sorted(). Include example code snippets.

References

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

2. Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.

3. Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). Green Tea Press.

4. Beazley, D., & Jones, B. K. (2013). *Python Cookbook* (3rd ed.). O'Reilly Media.

Suggested Reading

1. Beazley, D., & Jones, B. K. (2013). *Python Cookbook* (3rd ed.). O'Reilly Media.

2. VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media.

Web Resources

1. Python Software Foundation. (n.d.). *The Python Tutorial*. https://docs.python.org/3/tutorial/

2. W3Schools. (n.d.). *Python Tutorial*. https://www.w3schools.com/python/

Unit 3
Modules and Packages

Learning Outcomes

After completing this section, learners will be able to:

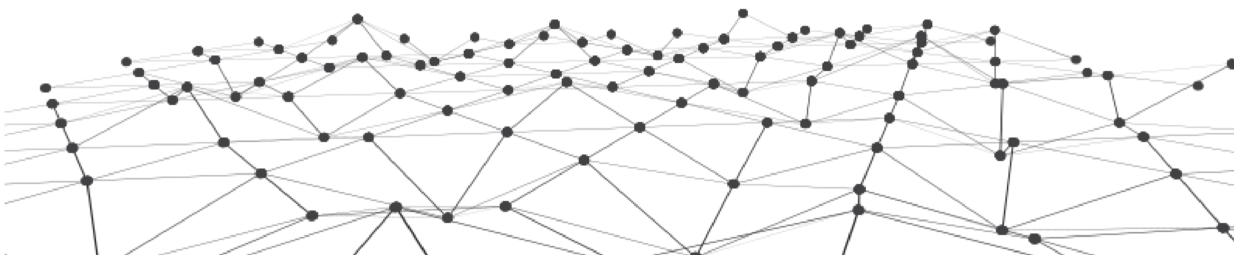
- define modules and packages in Python.
- identify different types of build-in modules in Python.
- describe the process of importing modules.
- explain how to describe a custom module and how Python finds and loads it during execution.
- summarize the structure and purpose of Python packages and how *__init__.py* is used.

Prerequisites

Before exploring modules and packages, it's important that you understand basic Python concepts such as variables, loops, functions, and conditional statements. These form the building blocks of any Python program. But as your projects grow larger like building a simple game or managing a library of books you'll quickly find that putting all your code into a single file becomes messy and hard to manage. That's when you need a better way to organize and reuse your code exactly what this unit offers.

Imagine you're building an online shopping app. You might have one part of the code that handles user login, another for displaying products, and another that processes payments. Writing all of that in one file would be confusing and hard to maintain. Instead, by using modules and packages, you can separate the code into logical parts like ***login.py, products.py, and checkout.py*** and then import only what you need. This approach makes the code cleaner, easier to understand, and more efficient to debug or upgrade just like professional software developers do.

Python's strength lies in its ability to let you write clean, reusable, and scalable code. This unit introduces you to importing existing modules like math and random, creating your own custom modules, and organizing them into packages. Learning this not only saves time but also prepares you for working on real-world applications whether you're developing games, automating tasks, analyzing data, or building websites. By master-



ing modules and packages, you're taking a major step from writing simple scripts to building powerful, well-structured programs.

Key words

Import, Variables, Custom Modules, Built-in Modules, Subpackages.

Discussion

3.3.1 Introduction to Python Modules

Python modules are like ready-made toolboxes that hold useful tools (code) you can use anytime in your programs. For example, imagine you have a box full of different colored crayons to draw pictures you don't have to make new crayons every time you want to color something. In Python, if you want to do math calculations, you can use the built-in `math` module, which already has many useful math functions like `sqrt()` for square roots. This way, you save time and keep your code neat by using modules instead of writing everything from scratch.

Python modules are files that contain Python code, defining functions, classes, and variables that can be utilized in other Python programs. Their purpose is to organize and reuse code, encouraging modularity and reusability. Modules aid in separating different concerns and making code easier to maintain. They can be either built-in modules included in the Python standard library or external modules developed by the Python community and installed using tools like pip. By importing modules into our programs, we can access their functionality and utilize their defined objects to perform various tasks, saving time and effort by avoiding the need to write code from scratch.

Using Python modules offers many benefits that make programming easier and more efficient. Modules help organize code into smaller, reusable parts, which improves clarity and maintainability. They also allow you to use pre-written code, saving time and effort. Overall, modules make it simpler to build, update, and manage Python programs, especially as projects grow larger and more complex. Some key advantages of Python modules are:

1. **Code Reusability:** You can write a piece of code once in a module and reuse it across many programs, saving time and effort.

2. **Better Organization:** Modules help break large programs into smaller, manageable files, making the code easier to understand and maintain.

3. **Namespace Management:** Modules provide separate namespaces, reducing the risk of name conflicts between variables and functions in different parts of a program.

4. **Simplified Maintenance:** When code is organized into modules, fixing bugs or updating features becomes easier because you only need to change the code in one place.

5. **Access to Built-in Functionality:** Python's rich standard library includes many useful built-in modules, so you can perform complex tasks without writing code from scratch.

6. **Collaboration Friendly:** Modules allow multiple developers to work on different parts of a project independently, improving teamwork and efficiency.

3.3.1.1 Creating a Python module (Creating custom modules)

Creating a Python module follows a simple syntax. To define a module, you create a new Python file with a ***.py*** extension. Inside this file, you can define functions, classes, and variables that you intend to use in other Python programs. These definitions allow you to organize and reuse code effectively.

Example:

Here's an example of creating a Python module named ***math_operations.py***:

```
# math_operations.py

        def add(a, b):

            return a + b

        def subtract(a, b):

            return a - b

        def multiply(a, b):

            return a * b

        def divide(a, b):
```

```
    if b != 0:
        return a / b
    else:
        print("Error: Division by zero is not allowed.")
```

In this example, the module ***math_operations*** contain four functions: add, subtract, multiply, and divide. These functions perform basic mathematical operations and can be reused in other Python programs.

3.3.1.2 The import Statement

The import statement is used in Python to bring modules or specific objects from modules into the current program's namespace. It allows us to access and utilize the functionality defined within the imported modules. This helps organize code and reuse functionality without rewriting it.

Syntax:

```
import module_name
```

Example	**Output**

import math_operations
 	
result = math_operations.add(5, 3)
 	
print(result)
	8

import math_operations
 	
result = math_operations.divide(10, 2)
 	
print(result)
	5.0

In the above code, we import the math_operations module and use its functions add and divide to perform addition and division operations respectively. This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the ***dot(.) operator*** is used.

3.3.1.3 Variables in Module

Variables within a Python module can be accessed and used by other programs that import the module. They serve as containers for storing data that can be shared between different parts of a program or even across multiple programs. This allows for efficient data organization and sharing, promoting code modularity and reusability.

Example:

# my_module.py

```
my_variable = “Hello, World!”

def print_variable():

print(my_variable)
```

In this example, the module ***my_module*** contains a variable named ***my_variable*** assigned with the string value “Hello, World!”. It also includes a function ***print_variable()*** that prints the value of ***my_variable.***

3.3.1.4 Naming a Module

In Python, a **module** is simply a file containing Python code, and the **name of the module** is the name of the file (without the `.py` extension). Choosing a clear and meaningful name for your module is important because it helps make your code easy to read, maintain, and reuse. Module names should follow standard naming rules use lowercase letters, avoid special characters, and keep the name short but descriptive. For example, a module that handles student data might be named `student.py`. Good naming practices improve the overall organization and structure of your Python programs. Choosing a suitable name for a Python module is crucial and involves the following guidelines:

1. **Descriptive:** Opt for a name that precisely describes the module’s purpose and functionality. A descriptive name allows others to understand its role with minimal effort.

2. **Concise:** Keep the module name short and avoid unnecessary length. Shorter names are easier to type, remember, and fit well within code.

3. **Lowercase:** Use lowercase letters for module names. This is a common convention in Python, distinguishing modules from classes and constants.

4. **Underscores:** When using multiple words in the module name, separate them with underscores(_) for better readability. For instance, favor “my_module” over “mymodule”.

5. **Avoid conflicts:** Ensure that the module name doesn’t conflict with any Python keywords or built-in module names. This helps prevent naming clashes and potential issues.

6. **Meaningful and self-explanatory:** Select a module name that conveys meaning and is self explanatory. This empowers developers, including yourself in the future, to grasp the module’s purpose without delving into the code details.

For example, if developing a module for string manipulation utilities, consider names like “string_utils” or “str_helpers”. These names clearly convey the module’s focus on string-related functionalities.

3.3.1.5 Renaming a Module

In Python, **renaming a module** means giving it a different name (alias) while importing it. This is done using the `as` keyword in the import statement. Renaming can make long or complex module names easier to type and read in your code. For example, instead of writing `import pandas`, you can write `import pandas as pd` and use `pd` throughout your program. This technique is especially helpful when working with commonly used libraries or when avoiding name conflicts between modules. Renaming does not change the actual file name - it only changes how you refer to it in your script. To rename a module in Python, you need to perform the following steps:

1. **Change the module file name:** Modify the name of the module file by renaming it while preserving the .py extension. For instance, if the original module file was named "old_module.py", rename it to "new_module.py".

2. **Update import statements:** Scan through other code files and locate import statements that refer to the original module name. Update these import statements to use the new module name instead. Replace occurrences of "import old_module" with "import new_module".

3. **Modify module references:** If there are any references to the original module within the code files, update them to reflect the new module name. For example, if there was a function called "old_module.some_function()", change it to "new_module.some_function()".

4. **Test and validate:** Execute the code and ensure that everything functions correctly after renaming the module. Verify for any errors or unexpected behavior, and make any necessary adjustments.

3.3.1.6 Executing a Module as a Script

To execute a Python module as a script, you can utilize the special construct if __name__ == "__main__". This construct allows you to differentiate between when the module is being directly executed as a script versus when it is being imported as a module.

Here's an example of how to execute a module as a script:

```
def add(a, b):

 return a + b

def subtract(a, b):

 return a - b

if __name__ == "__main__":

 # Code block executed when the module is run as a script

          result = add(5, 3)

 print("Result:", result)
```

In the above example, the module defines two functions, add and subtract. The if __ name__ == "__main__" condition is used to determine if the module is being directly executed. If it is, the code block within the condition will be executed.

To run the module as a script, you can execute the following command in the terminal or command prompt:

python module_name.py

Replace module_name with the actual name of your module file. This will execute the code within the if __name__ == "__main__" condition.

When the module is imported and used by another Python script, the code within the if __name__ == "__main__" block will not be executed. This allows the module to be imported and its functions to be used without any interference from the script-execution-specific code.

By using the if __name__ == "__main__" construct, you can execute specific code when running a module as a script while still enabling it to be imported and utilized as a module in other scripts.

3.3.1.7 The Module Search Path

The module search path in Python is a collection of directories that the Python interpreter examines when attempting to import modules in a program. The sys.path variable, which is a list of directory locations, determines this search path.

When importing a module, Python follows a specific order to search for the module in different locations. The search path is checked in the following sequence:

- **The current directory:** Python checks the directory from which the script or interactive interpreter is executed. This allows for importing modules from the same directory as the script.
- **PYTHONPATH environment variable:** Python examines the directories specified in the PYTHONPATH environment variable. This variable contains a list of directory names separated by colons (on Unix-like systems) or semicolons (on Windows systems).
- **Default module directory:** Python checks the standard library directories that are part of the Python installation. These directories contain built-in modules and other standard library modules.
- **Third-party module directories:** If the module is not found in the previous locations, Python searches in directories typically used for installing third-party modules. These directories are commonly determined by package managers, such as site-packages or dist-packages.

The module search path can be modified programmatically by adding or modifying entries in the sys.path list. This can be helpful when you need to include additional directories for module searching during runtime.

To examine the current module search path, you can access the sys.path variable:

import sys

print(sys.path)

This will display a list of directories constituting the module search path. dir() function In Python, the dir() function is a powerful tool for inspecting the contents of a module and retrieving a list of names, attributes, and methods defined within it. By calling dir(module_name), you can explore the specific names associated with that module.

Here's an example demonstrating the usage of dir() on a module:

# Import a module

```
        import my_module
```

# Display names, attributes, and methods of the module

```
print(dir(my_module))
```

In the above code, we import the my_module module and then use the dir() function to retrieve a list of names associated with it. The output will include functions, variables, classes, and other objects defined within the my_module.

Using dir() on a module provides valuable insights into the available functionality and objects within the module. It helps in understanding what the module offers and allows you to utilize its attributes and methods effectively.

Keep in mind that the dir() function provides only the names defined within the module, without providing detailed explanations or documentation. For more information about a specific attribute or method, you can use the help() function, passing the module and the name as arguments (e.g., help(my_module.some_function)).

By leveraging the dir() function, you can explore the contents of a module and harness its capabilities to build robust and efficient Python programs.

3.3.1.8 Built-in Modules

In Python, **built-in modules** are special code libraries that come **pre-installed** with Python. These modules contain ready-made functions and tools that help you perform common programming tasks such as math calculations, working with dates, generating random numbers, or interacting with the operating system. Instead of writing these functions yourself, you can simply use the `import` statement to bring in a built-in module and start using its features right away. This makes your code **shorter, cleaner, and more efficient**, especially when solving everyday problems in Python. Some commonly used built-in modules in Python include "math" for mathematical operations, "random" for random number generation and selection, "datetime" for manipulating and formatting dates and times, "os" for performing operating system-related tasks, "sys" for system-specific operations, "re" for working with regular expressions, "json" for JSON manipulation, "csv" for handling CSV files, "urllib" for working with URLs and HTTP operations, and "sqlite3" for interacting with SQLite databases (Table 3.3.1).

Table 3.3.1 Build-in modules in Python

Module Name	**What It Does**	**Example**

math	Math functions like square root, sin	import math
		math.sqrt(25)

random	Generate random numbers	import random
		random.randint(1, 10)

datetime	Work with dates and time	import datetime
		datetime.date.today()

os	Work with the operating system	import os
		os.getcwd()

sys	Get info about the Python environment	import sys sys.version

re	Work with regular expressions (text patterns)	`import re`
		`re.search(r"\d+", "Age: 25")`

urllib	Handle URLs and web requests	`import urllib`
		`urllib.request.urlopen("https://example.com")`

json	Work with JSON data	`import urllib`
		`json.dumps({"x": 1})`

csv	Work with CSV files	import csv
		csv.reader(open("data.csv"))

These built-in modules offer a wide range of functionalities and simplify common programming tasks, providing developers with efficient and effective tools for their Python programs.

3.3.2 Packages

Python packages serve as a means to structure and distribute Python code effectively. Essentially, a package is a directory containing one or more Python modules, along with an optional special file named __init__.py. It facilitates the grouping of related modules, establishing a hierarchical organization for your code.

The primary purpose of packages is to enable code organization and reuse in a modular and scalable manner. By organizing modules into packages, you can prevent naming conflicts, enhance code maintainability, and improve code readability. Moreover, packages can be shared with others, fostering code collaboration and reuse.

3.3.2.1 Creating Packages

To create a package, you create a directory with a unique name and include the __init__.py file within it. The __init__.py file can either be left empty or contain initialization code that executes when the package is imported. Packages can have sub-packages, which are essentially nested directories containing their own _init__.py files. This nesting capability allows for a hierarchical arrangement of packages, facilitating the organization of code at different levels of abstraction.

Package installation and management can be handled using package managers like pip, which is the predominant package manager in the Python ecosystem. Utilizing pip, you can effortlessly install, upgrade, and uninstall packages from the Python Package Index (PyPI), a community-maintained repository of Python packages.

Once a package is installed, its modules can be imported and utilized in other Python scripts through the import statement. This statement grants access to functions, classes, and variables defined within the package's modules.

3.3.2.2 Package Initialization

Package initialization in Python refers to the process of preparing a package for use when it is imported. It involves executing the code within the __init__.py file located in the package directory.

The __init__.py file serves as an indicator that the directory is a Python package. It can contain Python code that is executed during package import. This initialization code typically handles tasks like importing specific modules, setting up package-level variables, or performing any necessary initialization logic. Common scenarios for package initialization include:

- **Importing Modules:** The __init__.py file can include import statements to bring in modules within the package. This simplifies access to the package's modules and their contents when the package is imported.
- **Setting Package-Level Variables:** Initialization code can define variables that are accessible at the package level. These variables can be shared among the package's modules or used for configuration purposes.
- **Executing Initialization Logic:** The __init__.py file can contain code that carries out initialization steps required by the package. This may involve tasks such as establishing database connections, configuring logging, or registering components.

While the __init__.py file is optional, it provides a way to customize package behavior during import and serves as a central location for package initialization.

When a package is imported, the Python interpreter automatically executes the code within the __init__.py file, if present. This initialization code runs only once, regardless of how many times the package is imported in a program.

By leveraging package initialization, you ensure that your package is properly set up and ready for use upon import. This simplifies package organization and allows for better control over the package's behavior and functionality.

Example:

Suppose you have a package named "my_package" with the following directory structure:

```
my_package/

__init__.py

 module1.py

 module2.py
```

To initialize the package, follow these steps:

Create the __init__.py file: Inside the "my_package" directory, create a file named __init__.py. This file can be left empty or include initialization code.

Define module files: Within the "my_package" directory, create two Python module files, module1.py and module2.py. Each module will contain functions or variables related to specific functionalities.

Here's an example of how you can initialize the package:

```
init.py:

print("Initializing my_package...")

# Import modules within the package

from . import module1

from . import module2

module1.py:

def function1():

print("This is function 1 from module 1")

module2.py:

def function2():

print("This is function 2 from module 2")
```

In the __init__.py file, we print a message to indicate that the package is being initialized. We also import the modules module1 and module2 using relative imports (from . import ...).

Now, let's use the package in another Python script:

```
main.py:

import my_package

print("Package imported!")

my_package.module1.function1()

my_package.module2.function2()
```

When you run the main.py script, the output will be:

```
Initializing my_package...

Package imported!

This is function 1 from module 1

This is function 2 from module 2
```

In this example, when the my_package package is imported, the __init__.py file is executed, and the initialization code within it is run. It prints the initialization message and imports the module1 and module2 modules. You can then access the functions defined in the modules using the package name and module name as demonstrated in main.py. This example demonstrates the process of package initialization in Python, where the __init__.py file is crucial for setting up the package during import. Remember to modify the package and module names and customize the functionality based on your specific requirements.

3.3.2.3 Subpackages

Subpackages in Python provide a means of structuring and organizing code within packages in a hierarchical manner. They facilitate improved modularity, organization, and reusability of related components or functionality. Subpackages enable the creation of complex projects by establishing a multi-level package structure. To create a subpackage, you can follow these steps:

- Begin by creating the main package directory, which serves as the parent directory for the subpackage. This directory should include an init.py file that can either be left empty or contain initialization code specific to the package.
- Inside the main package directory, create a subdirectory with a unique name that will function as the subpackage. This subdirectory should also have an init.py file, which can be empty or contain initialization code specific to the subpackage.

- Include one or more module files (Python files) within the subpackage directory to hold the code relevant to the subpackage. These modules can consist of functions, classes, or other code elements.

By utilizing subpackages, developers can enhance code maintainability, separation of concerns, and code reuse. Subpackages allow for the logical grouping of related modules, facilitating easier navigation and utilization of specific functionality within a project.

Example: Suppose you are working on a project related to geometry calculations and want to organize your code into subpackages. You can create a main package called "geometry" and include subpackages such as "shapes" and "utils".

The directory structure would look like this:

```
geometry/
 __init__.py
 shapes/
 __init__.py
 circle.py
 rectangle.py
 utils/
 __init__.py
 calculations.py
```

In this example, the "geometry" package serves as the main package. It contains two subpackages, "shapes" and "utils", represented by separate directories. Each subpackage has its own __init__.py file, indicating that they are Python subpackages. The "shapes" subpackage includes two module files: "circle.py" and "rectangle.py". These files can contain classes and functions related to calculations and properties of circles and rectangles. The "utils" subpackage consists of one module file: "calculations.py". This file can contain utility functions or calculations that are commonly used in geometry operations.

To import and use modules from the subpackages, you can use the dot notation: from geometry.shapes.circle import Circle from geometry.utils.calculations.

```
import calculate_area
circle = Circle(radius=5)
area = calculate_area(circle)
print("The area of the circle is: {area}")
```

Here, we import the Circle class from the "shapes.circle" subpackage and the calculate_ area function from the "utils.calculations" subpackage. This allows us to create a circle object and calculate its area using the imported functionality.

This unit focuses on organizing Python code through modular programming. It begins by explaining how to use the `import` statement to access built-in and user-defined modules, helping to reduce code duplication and improve clarity. Learners then explore how to create their own modules by writing reusable functions and classes in separate `.py` files. The unit also introduces packages, which are collections of modules grouped in directories, typically containing an `__init__.py` file to signal Python that the folder is a package. Together, these concepts help students write cleaner, more manageable, and scalable Python programs by breaking complex code into simpler, reusable parts.

Objective Type Questions

1. What does init.py file do?
2. What keyword is used to import specific items from a module?
3. What keyword is used to create a package?
4. What is a module in Python?
5. Which keyword is used to import a module?
6. Name a built-in Python module.
7. What is the file extension of a Python module?
8. What does from math import sqrt do?
9. What will import math; print(math.pi) output?
10. How do you import all contents from a module?
11. Which function lists all functions and attributes in a module?
12. How do you create a custom module?
13. What is a package in Python?
14. What does the __init__.py file do in a package?
15. How do you import a module from a package?
16. What does dir(math) return?
17. Which module is used for working with URLs?
18. Which built-in module helps interact with the operating system?
19. What does sys.path contain?
20. What is a subpackage?

Answers to Objective Type Questions

1. Initialization
2. from
3. init.py
4. A file containing Python code such as functions, classes, or variables.
5. import
6. math, json, random
7. .py
8. It imports only the sqrt function from the math module.
9. 3.141592653589793
10. from module import *
11. dir() function
12. Create a .py file containing functions or classes.
13. A directory containing Python modules and an __init__.py file.
14. It marks the directory as a Python package.
15. Using import package.module or from package import module
16. A list of all attributes and functions available in the math module.
17. urllib
18. os module
19. A list of directories Python searches for modules.
20. A package inside another package.

Assignments

1. Create a module that contains functions for calculating the area and circumference of a circle, and import it to calculate these values for user-provided input.

2. Design a package with subpackages representing different categories of animals, each containing modules with functions to display information about specific animals, and import them to display details based on user input.

3. Create a module that includes a function to generate a random password, and import it to generate and display a password with a user-defined length.

4. Develop a package with subpackages for basic mathematical operations (addition, subtraction, etc.) and import the appropriate subpackage and module to perform calculations based on user input.

Reference

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

2. Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.

3. Beazley, D., & Jones, B. K. (2013). *Python Cookbook* (3rd ed.). O'Reilly Media.

4. Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). Green Tea Press.

5. Pilgrim, M. (2009). *Dive Into Python 3*. Apress.

Suggested Reading

1. Python Official Documentation. *Modules*. https://docs.python.org/3/tutorial/modules.html

2. Real Python. *Python Modules and Packages: An Introduction*. https://realpython.com/python-modules-packages/

3. GeeksforGeeks. *Python Modules and Packages*. https://www.geeksforgeeks.org/python-modules-packages/

4. W3Schools. *Python Modules*. https://www.w3schools.com/python/python_modules.asp

5. Programiz. *Python Modules and Packages*. https://www.programiz.com/python-programming/modules-packages

Unit 4
Regular Expression

Learning Outcomes

After completing this section, learners will be able to:

- familiarize the concept of regular expressions and their uses.
- identify and use special sequences in regex.
- utilize the re module functions such as match(), search(), findall(), and sub().
- apply regex for searching, matching, and replacing patterns in text.
- differentiate between string methods and regex methods.

Prerequisites

Regular expressions are important because they help us quickly search, match, or replace patterns in text. In real-world situations like checking if an email is valid, finding phone numbers in a file, or removing unwanted spaces in a document regex makes the job faster and more accurate. Instead of writing long and complex code, a single regex pattern can do the work efficiently.

To understand regular expressions, it's helpful to know basic Python topics such as working with strings, using loops and conditions, and simple functions like replace() and split(). These skills make it easier to learn how regex patterns behave and how they interact with text using Python's re module.

Key words

Pattern Matching, Special Sequence, Python ReModule, Search Function, Replace Function, Text Processing

Discussion

3.4.1 Understanding regex patterns

A Regular Expression (or regex) is a special pattern used to search, match, or find specific parts of text. It's like giving the computer instructions on what kind of text you are looking for. For example, if you want to find all phone numbers in a file or check if an email address is correct, regex can help.

In order to find a string or group of strings, a Regular Expression (RegEx) is a unique string of characters. By comparing a text to a specific pattern, it may determine if it is present or absent. It can also divide a pattern into one or more sub-patterns. Regex functionality is available in Python through the re module. Its main purpose is to provide a search, for which a string and a regular expression are required. It either returns the first match in this case or none at all.

3.4.1.1 Special Sequences

A special sequence is a '\' followed by one of the characters in the list below, and has a special meaning:

Table 3.4.1 Special Sequences

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"

\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"

\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"

3.4.2 re Module

In Python, the re module stands for "regular expressions". This module provides powerful tools to search, match, and manipulate strings using patterns. Think of it as a smart search tool that helps you find things in text, like finding phone numbers in a document, checking if an email is valid, or replacing specific characters. Unlike simple string functions, the re module allows you to define complex rules using special characters and patterns.

To use it in your Python program, you need to import it first:

import re

Once imported, you can use its various functions to work with patterns and strings.

Use of re Module

Suppose you want to extract all phone numbers from a paragraph, or remove extra spaces, or hide sensitive data like email addresses. Doing these tasks with basic string methods (replace, split, etc.) would require long, repetitive code. But with re, you can do it with just one line using patterns. The re module makes these jobs easier, faster, and more accurate.

Common Functions in the re Module

Here are the main functions provided by the re module:

Table 3.4.2 Common functions in the re Module

Function	Purpose
re.match()	Checks if the **pattern matches from the beginning** of the string.
re.search()	Searches the **entire string** and returns the **first match**.
re.findall()	Returns a **list of all non-overlapping matches**.
re.sub()	**Replaces** parts of the string that match the pattern.
re.split()	**Splits** a string based on a pattern (like a smart version of split()).
re.fullmatch()	Checks if the **entire string matches** the pattern.

3.4.3 Searching

Once you understand how regex patterns work, the next important step is learning how to search for those patterns inside a piece of text. In Python, this is done using the re module, which provides a function called re.search() (already mentioned in table 3.4.2) . This function is used to scan through a string and check if the pattern exists anywhere in the text. If the pattern is found, it returns a match object, which contains information about what was found and where it was found. If the pattern is not found, it returns None, which means there is no match.

Let's say you have a sentence like "The cat is sleeping." and you want to check if the word "cat" is in it. You can create a regex pattern cat and use it with re.search() to see if that pattern appears in the text.

Example 1:

```
import re

text = "The cat is sleeping."

pattern = r"cat"

match = re.search(pattern, text)

if match:

    print("Found:", match.group())
```

This will output **Found: cat** because the pattern cat is present in the string.

You can also use regex to search for more complex things. For example, if you want to check if there is a number in the text, you can use the pattern \d+. Here, \d means "a digit", and + means "one or more times". So this pattern will match numbers like 3, 45, or 6789.

Example 2:

```
text = “Order number is 12345.”

pattern = r”\d+”

match = re.search(pattern, text)

if match:

    print(“Found number:”, match.group())
```

This will output, **Found number: 12345** because the number appears in the string.

It’s important to note that re.search() only finds the first occurrence of the pattern. Even if the pattern appears multiple times, it will only return the first match. For example, in the sentence “There are 2 cats, 3 dogs, and 5 birds,” if you search using the pattern \d+, it will only return 2, even though there are three numbers in the sentence.

Example 3:

```
text = “There are 2 cats, 3 dogs, and 5 birds.”

pattern = r”\d+”

match = re.search(pattern, text)

print(“First number found:”, match.group())
```

This will output, **First number found: 2.**

You can also search for patterns at the beginning or end of a string using special symbols. The caret symbol ^ is used to match text at the start, and the dollar symbol $ is used to match text at the end. For instance, if a sentence starts with the word “Hello”, the pattern ^Hello will match. Similarly, if a sentence ends with the word “end.”, the pattern end\.$ will match.

Example 4:

```
text = “Hello, how are you?”

pattern = r”^Hello”

match = re.search(pattern, text)

print(bool(match))
```

Output

```
True
```

Example 5:

```
text = “This is the end.”

pattern = r”end\.$”

match = re.search(pattern, text)

print(bool(match))
```

Output

```
True
```

In both cases, the patterns match because the words appear exactly where expected at the beginning or the end. Using re.search() is very helpful for checking whether something exists in text, such as a name, keyword, number, or format. Once a match is found, you can use .group() to get the matched text, or use it in programs to make decisions. Although re.search() only returns the first match, it is a very powerful way to detect patterns quickly in strings of any size.

3.4.4 Matching

Matching with regular expressions is similar to searching, but with a more specific purpose. In Python, when you want to check if a string starts exactly with a certain pattern, you use the re.match() function. While re.search() looks for a pattern anywhere in the string, re.match() checks only at the very beginning of the string. If the pattern appears at the start, it returns a match object; otherwise, it returns None.

Imagine you have the sentence “Hello world”. If you use re.match() with the pattern Hello, it will return a match because “Hello” is right at the beginning. But if you try to match the word “world”, it will return nothing, because “world” is not at the start.

Example 1: Match at the beginning

```
import re

text = “Hello world”

pattern = r”Hello”

match = re.match(pattern, text)

if match:

    print(“Matched:”, match.group())

 # Output: Matched: Hello
```

Now let’s try matching a word that is not at the beginning.

Example 2: No match

```
text = “Hello world”
```

```
pattern = r"world"

match = re.match(pattern, text)

print(match)

 # Output: None
```

So, re.match() is strict. It only matches if the pattern appears right from the start of the string. If you want to find a match anywhere in the string, even in the middle or end, you should use re.search() instead.

Another useful function is re.findall(). This function returns all matches of a pattern in the string, not just the first one. It gives the result as a list of matched items. For example, if you want to find all numbers in a sentence, you can use the pattern \d+ with re.findall().

Example 3: Find all matches

```
text = "There are 2 cats, 3 dogs, and 5 birds."

pattern = r"\d+"

matches = re.findall(pattern, text)

print("Numbers found:", matches)

# Output: ['2', '3', '5']
```

re.findall is very useful when you want to extract every instance of a certain pattern, like all numbers, emails, or words from a paragraph.

There's also another function called re.finditer() which is similar to findall(), but instead of returning just the matched strings, it gives you match objects for each result. This is helpful when you want to know where in the text each match was found.

Example 4: Using re.finditer()

```
text = "There are 2 cats and 3 dogs."
pattern = r"\d+"
matches = re.finditer(pattern, text)
for match in matches:
    print("Found:", match.group(), "at position", match.start())

# Output:
Found: 2 at position 10
Found: 3 at position 21
```

String	T	h	e	r	e		a	r	e		2		c	a	t	s		a	n	d		3		d	o	g	s	.
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

↑ 2 at pos 10

↑ 3 at pos 21

Fig 3.4.1 Visual Breakdown of text

This will print each number found and its position in the string. So while re.match() is best for checking if a string starts with something, re.findall() and re.finditer() are better when you want to find all occurrences of a pattern in a string.

Matching with regex is very helpful when you're processing large amounts of text and need to find specific parts quickly and accurately. Whether you want to match a word at the beginning, find every email address, or extract all numbers from a document, these regex functions can help you do that with just a few lines of code.

3.4.5 Replacing text

When we work with text in Python, we often come across situations where we want to replace some words or characters with something else. For example, imagine you're editing a document and you want to replace every occurrence of the word "cat" with "dog". Doing it manually is time-consuming, especially with large files. Luckily, Python provides simple ways to do this using string methods and regular expressions.

- Simple Replacement Using str.replace()

Python has a built-in method called replace() that allows you to replace a specific piece of text with another.

Example:

```
text = "I have a red car"

new_text = text.replace("red", "blue")

print(new_text)

# Output: I have a blue car
```

Explanation: The word "red" was replaced with "blue". This is a direct and easy way to replace exact words. However, it cannot handle patterns or rules, like "replace all numbers" or "remove extra spaces".

- Replacing Using Regular Expressions (Regex)

To replace text based on patterns (like replacing all digits, special characters, or email addresses), we use Python's re module and its powerful function: re.sub().

Syntax: re.sub(pattern, replacement, text)

Pattern: the regex pattern you want to find.

Replacement: the new text you want to insert.

text: the original string where replacement will happen.

Example 1: Replace All Digits with a Symbol

Let's say you have a sentence that contains some numbers, and you want to replace all numbers with #.

```
import re

text = “My phone number is 9876543210”

new_text = re.sub(r’\d’, ‘#’, text)

print(new_text)

# Output: My phone number is ##########
```

Explanation: \d is a regex pattern that matches any digit (0–9). Every digit in the sentence is replaced with a #.

Example 2: Remove Extra Spaces

Sometimes, text has extra spaces that you want to clean up.

```
text = “This    sentence   has  too  many   spaces.”

clean_text = re.sub(r’\s+’, ‘ ‘, text)

print(clean_text)

# Output: This sentence has too many spaces.
```

Explanation: \s+ means “one or more spaces”. It replaces multiple spaces with a single space.

Example 3: Hide Email Addresses

In a document, you might want to hide email addresses for privacy.

```
text = “Contact me at john.doe@example.com”

hidden = re.sub(r’\S+@\S+’, ‘[email hidden]’, text)

print(hidden)

#Output: Contact me at [email hidden]
```

Explanation: \S+@\S+ matches any pattern that looks like an email address. It replaces the email with [email hidden].

Example 4: Replace Only First Match

You can also replace only the first match using an optional parameter.

```
text = "apple apple apple"

new_text = re.sub(r'apple', 'orange', text, count=1)

print(new_text)

#Output: orange apple apple
```

Explanation: Only the first "apple" is replaced with "orange".

Use str.replace() when you know the exact text, and re.sub() when you need more power and flexibility.

Recap

- ♦ Regex is used for pattern matching in strings.
- ♦ Special sequences like \d, \w, \s help target specific character types.
- ♦ The re module enables advanced string operations.
- ♦ re.search() finds the first match anywhere.
- ♦ re.match() only checks at the beginning.
- ♦ re.findall() returns all matches.
- ♦ re.sub() replaces patterns with another string.
- ♦ Regex simplifies complex string tasks like data cleaning.

Objective Type Questions

1. What function returns all pattern matches in a string?
2. What does \d match?
3. Which regex function checks only the beginning of a string?
4. What operator is used for string concatenation in Python?
5. What function replaces text based on a regex pattern?
6. What function returns match objects with their positions?
7. Which method should you use to remove extra spaces from a string?

8. What does \s represent in regex?
9. Which regex method substitutes part of a string?
10. What regex character is used to represent any character except a newline?
11. What function is used to search a pattern in a string?
12. What symbol is used to indicate the start of a string in regex?
13. What symbol matches the end of a string?
14. Which special sequence matches a word character?
15. Which regex symbol means zero or more repetitions?

Answers to Objective Type Questions

1. re.findall()
2. A digit character ([0-9])
3. re.match()
4. + (plus operator)
5. re.sub()
6. re.finditer()
7. str.strip() *(or)* re.sub(r'\s+', ' ', text).strip()
8. Any whitespace character
9. re.sub()
10. . (dot)
11. re.search()
12. ^
13. $
14. \w
15. *

Assignments

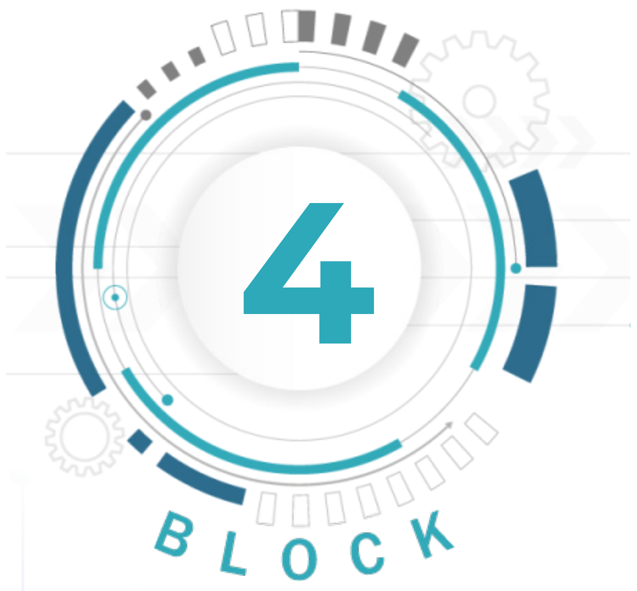
1. Write a Python program to find all email addresses in a string using regex.
2. Replace all digits in the given sentence with *.
3. Explain the difference between re.match() and re.search() with examples.
4. Extract all numbers from: "The 3 cats have 2 toys and 5 balls."
5. Replace the first occurrence of "apple" with "orange" using re.sub().

Reference

1. Matthes, E. (2023). *Python crash course: A hands-on, project-based introduction to programming* (3rd ed.). No Starch Press.
2. Sweigart, A. (2024). *Automate the boring stuff with Python: Practical programming for total beginners* (3rd ed.). No Starch Press.
3. Downey, A. B. (2023). *Think Python: How to think like a computer scientist* (2nd ed.). Green Tea Press.
4. Zelle, J. M. (2022). *Python programming: An introduction to computer science* (3rd ed.). Franklin, Beedle & Associates.

Suggested Reading

1. Python Software Foundation. (2023). *Python documentation: re module*. https://docs.python.org/3/library/re.html
2. Barry, P. (2022). *Head First Python: A brain-friendly guide* (3rd ed.). O'Reilly Media.



File Handling and Object-Oriented Programming

Unit 1

Introduction to File Handling

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- identify different file modes used in Python file operations.
- familiarize with key file methods used for reading, writing, updating, and managing files.
- explore the use of the with statement to manage files safely and efficiently.

Prerequisites

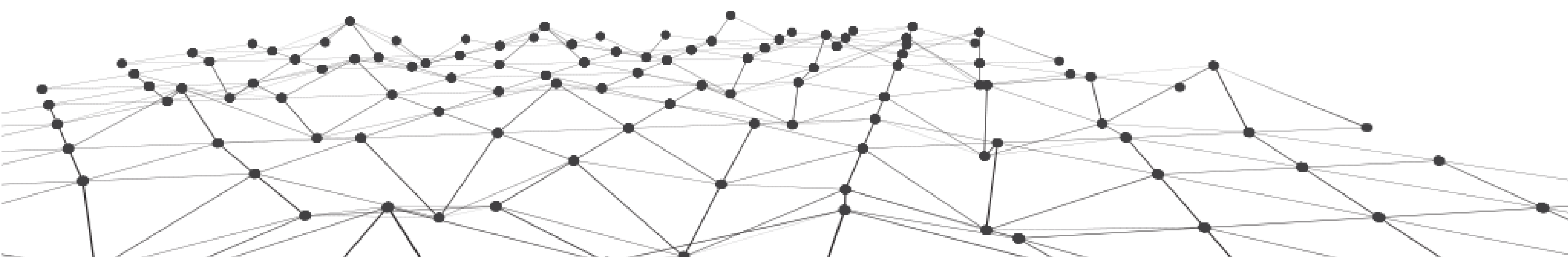
Long before modern programming languages, computers processed data in real time. Once the program stopped, all the data disappeared. There was no way to store results, save inputs, or revisit outputs. To overcome this limitation, the idea of **files** was introduced. A file became a convenient way to organize and store data on a storage device, making it possible to retrieve or update it even after a program ends. This shift transformed programming from temporary calculations to permanent record keeping.

Now, think about the digital world around you: music playlists, saved passwords, downloaded documents, and even browser history. These are all made possible because programs know how to handle files. Without file handling, every time you closed an app, all your preferences and progress would vanish. As programmers, learning how to work with files gives us the power to make software smarter, more interactive, and persistent.

In this lesson, you'll explore how Python, with just a few lines of code, can open, read, write, and modify files. Once you master this, your programs will no longer be forgetful - they'll be able to store and retrieve data like any modern application.

Key words

'r', 'w','a', 'r+', 'w+', 'x', 'rb', seek( ), with



Discussion

In real-world applications, data often needs to be stored and retrieved for future use. This includes tasks such as saving user information, configuration settings, or logging data. Python file handling provides a simple and efficient way to interact with files on your system. It allows you to create, read, write, and modify files using built-in functions. With just a few lines of code, Python can open files in different modes, append data, or overwrite content. This makes it a powerful tool for managing data in a persistent and organized manner. File handling is an essential skill for every Python programmer, especially when working with data processing or automation tasks.

Once you understand the importance of file handling, the next step is to learn how to perform basic operations such as opening a file, reading its content, writing new data, or updating existing data.

4.1.1 Opening a file in Python

To begin working with files in Python, it is important to learn how to open a file using the built-in open( ) function. This function allows the program to access a file stored on the system and perform actions such as reading, writing, or appending content. Depending on the requirement, the file can be opened in different modes, each serving a specific purpose. Once a file is opened, Python offers a simple and readable syntax to perform the desired operations efficiently.

Syntax

```
open("filename", "mode")
```

"filename" is the name of the file you want to work with.

"mode" specifies the purpose: read, write, append, etc.

When working with files in Python, it is important to specify how you want to interact with the file. You might want to read its contents, write new data, or add to what already exists. This is done using file modes, which are passed as a second argument to the open( ) function. Each mode serves a different purpose and affects how the file is accessed or modified. The common modes are :

1. "x" – Create (creates a new file and raises an error if it already exists)
2. "r" – Read (default mode)
3. "w" – Write (creates a new file or overwrites if it exists)
4. "a" – Append (adds content to the end of the file)

4.1.1.1 Creating a new File

To create a file only if it does not exist, use "x" mode.

Example:

```
f = open("newfile.txt", "x")
```

If newfile.txt already exists, Python will raise an error.

4.1.1.2 Reading from a File

Suppose we have a file named example.txt with some content:

Hello, Python learners!

Welcome to file handling.

Example:

We can read the content using:

```
with open("example.txt", "r") as f:
    data = f.read( )
    print(data)
```

Output :

```
Hello, Python learners!
Welcome to file handling.
```

Using with automatically closes the file after use.

4.1.1.3 Writing to a File

To write data to a file, use the "w" mode. If the file exists, it will be overwritten.

Example:

```
with open("example.txt", "w") as f:
    f.write("This is a new line of text.")
```

If you now read the file, the old content will be gone.

4.1.1.4 Appending to a File

To add content without deleting existing data, use "a" mode.

Example

```
with open("example.txt", "a") as f:
    f.write("\nThis line is added to the file.")
```

Output after appending:

```
This is a new line of text.
This line is added to the file.
```

4.1.1.5 'b' Binary mode

The binary mode is used when working with non-text files, such as images, audio files, executable programs, or any other type of file that contains binary data. When you open a file in binary mode, you're telling Python to handle the file as a sequence of bytes instead of text. This mode does not perform any encoding or decoding; what you read or write is exactly what exists in the file.

Binary mode is used in combination with other file modes:

- 'rb' → Read binary file
- 'wb' → Write binary file (overwrite)
- 'ab' → Append binary data

Example

```
with open("photo.jpg", "rb") as file:
        content = file.read( )
```

In this example, the image is read as raw byte data, which can be useful for processing or copying image files.

4.2.2 seek( ) Function in Python

The seek( ) function is used to change the current position of the file pointer within an open file. It allows you to move the file pointer to a specific location, which is especially useful when you need to read or write at a particular position in the file.

Syntax:

```
file_object.seek(offset, whence)
```

Where, offset is the number of bytes to move the file pointer. &

whence (optional): Specifies the reference position from where the offset is applied. It can be:

- 0 – Beginning of the file (default)
- 1 – Current file position
- 2 – End of the file

Example 1:

```
with open("example.txt", "r") as f:
         f.seek(5)
        data = f.read()
        print(data)
```

This example moves the file pointer 5 bytes from the beginning of the file before starting to read.

Example 2:

```
with open("example.txt", "r") as f:

        f.read(4)  # Read the first 4 bytes

        f.seek(3, 1)  # Move the file pointer 3 bytes forward from the current position

        data = f.read( )  # Read the remaining content from the new position

        print(data)
```

This example first reads 4 bytes, then skips 3 more bytes from the current position before reading again.

The r mode alone (i.e., "r" without binary) does not support whence=2 in the seek( ) function on all systems, especially in text mode. Use binary mode ("rb", "r+b" etc.) if you need to seek relative to the end of the file using whence=2.

4.1.3 Combined File modes

In Python, combined file modes provide the flexibility to perform both reading and writing operations within a single file access. Unlike basic modes that limit actions to either reading or writing, these combined modes enable more dynamic interactions with files, such as updating content, appending data while still being able to read it, or creating a new file with full access. Understanding how and when to use these combined modes is essential for efficient file manipulation in real-world applications.

Let's have an explanation of the combined file modes in Python, along with illustrative examples for each.

4.1.3.1 r+ mode – Read and Write

This mode opens the file for both reading and writing. The file must already exist, and the file pointer is positioned at the beginning.

Example

```
# Assuming 'sample.txt' contains: Hello World

with open("sample.txt", "r+") as f:

   content = f.read()

   print("Before write:", content)

   f.seek(0)  # Move the cursor to the beginning

   f.write("Hi")  # Overwrites 'He' in 'Hello'

Result in file: Hi llo World
```

4.1.3.2 w+ mode – Write and Read

Opens the file for both writing and reading. Content is **overwritten** if the file exists.

Example

```
        with open("sample.txt", "w+") as f:
                f.write("Python is fun!")
                f.seek(0)  # Move to the beginning to read
                print(f.read())
```

Result in file: Python is fun!

4.1.3.3 a+ mode – Append and Read (creates if not exists)

Opens the file for both reading and appending. Reading is possible, but writing always appends to the end.

Example

```
with open("sample.txt", "a+") as f:
        f.write("\nLet's learn file handling.")
        f.seek(0)  # Move to beginning to read entire content
        print(f.read())
```

Result in file: Original content + appended line

4.1.3.4 x+ mode – Create and Read/Write (fails if exists)

Creates a new file and opens it for reading and writing. If the file already exists, an error is raised.

Example

```
try:
                with open("newfile.txt", "x+") as f:
                        f.write("This is a new file.")
                         f.seek(0)
                        print(f.read())
except FileExistsError:
                print("File already exists.")
```

Output:

If file doesn't exist – creates it and prints content.

If file exists – prints: File already exists.

4.1.4 File closing in Python

When a file is opened using open( ), it occupies system resources. To release these resources, the file must be closed using the close( ) method. This is especially important when writing to a file, as it ensures that all buffered data is properly saved.

Why should we close files?

Closing a file after you're done with it might seem like a small thing, but it is actually very important.

- Freeing up system resources

When a file is opened, your computer uses memory and system resources to keep track of it. If you open too many files and forget to close them, your program or even the whole system could run into trouble. For example, it may run out of memory or hit a limit on how many files can be open at once. Closing the file tells your system that the job is done and the resources can be released.

- Making sure data is saved

Sometimes when you write data to a file, it is not saved immediately to the hard disk. It may be stored temporarily in memory. If you forget to close the file, that data might never actually be saved. Closing the file ensures that all the data is properly written and saved.

- Avoiding data problems

If your program crashes or shuts down before the file is properly closed, the file might end up incomplete or corrupted. Closing it right after you finish working with it helps protect your data from such problems.

4.1.4.1 File close( ) method

After working with a file (reading or writing), it is important to close the file properly using the close( ) method.

Syntax

file_object.close( )

Example : Closing a file after writing

```
f = open("demo.txt", "w")      # Open a file in write mode
f.write("This is a sample text.")       # Write some text to the file
f.close( )        # Close the file
```

Here, the file demo.txt is opened for writing. After writing text into it, we call f.close( ) to safely close the file and save the data.

Example : Closing a file after reading

```
f = open("demo.txt", "r")      # Open a file in read mode
```

```
content = f.read( )       # Read the contents
print(content)
f.close()          # Close the file
```

4.1.4.2 Using *with* statement

Instead of manually closing the file, Python provides the `with` statement which automatically closes the file after the block is executed:

```
with open("example.txt", "r") as f:
    data = f.read( )
# No need to call f.close()
```

4.1.5 File methods

1. file.fileno( ) – Returns the underlying file descriptor (an integer) used by the operating system to identify the open file.
2. file.seek(offset, whence=0) – Moves the file pointer to a specific position in the file, allowing random access to file content.
3. file.tell( ) – Returns the current position of the file pointer (in bytes) from the beginning of the file.
4. file.readline( ) – Reads and returns a single line from the file, including the newline character at the end.
5. file.truncate(size=None) – Resizes the file to the given size in bytes; if no size is specified, it truncates from the current position.

You will learn these methods in detail in the next unit.

A Sample program that demonstrates read, write and append modes.

```
# Step 1: Write initial content to the file
        f = open("notes.txt", "w")
        f.write("Day 1: Started learning Python.\n")
        f.close()
# Step 2: Append new content to the file
        f = open("notes.txt", "a")
        f.write("Day 2: Practiced file handling.\n")
        f.close()
```

```
# Step 3: Read and print the entire content
            f = open("notes.txt", "r")
            print("My Learning Notes:")
            print(f.read())
            f.close()
```

Output

```
My Learning Notes:
Day 1: Started learning Python.
Day 2: Practiced file handling.
```

Recap

Importance of File Handling

- Allows storing and retrieving data persistently
- Useful for saving user data, logs, and configuration files
- Essential for automating tasks involving data input/output

Basic Operations in File Handling

- Opening files using open()
- Reading and writing data
- Appending new content
- Modifying and deleting files

Syntax of open() Function

- open("filename", "mode")
  - "filename": Name of the file
  - "mode": Specifies the operation mode (read, write, etc.)

Common File Modes

- "r": Read mode, file must exist
- "w": Write mode, creates a new file or overwrites existing one
- "a": Append mode, adds data to the end of the file

- "x": Exclusive creation, error if file already exists
- "b": Binary mode, used for non-text files (like images)
- "t": Text mode, default for reading/writing text

Combined Modes

- "r+": Read and write, file must exist
- "w+": Write and read, file is created or overwritten
- "a+": Append and read, creates file if it doesn't exist
- "x+": Create and read/write, error if file exists

Using with Statement for Files

- with open("file.txt", "r") as f:
- Automatically closes the file after the block
- Safer and cleaner than using f.close() manually

Commonly Used File Methods

- read(): Reads entire file as a string
- readline(): Reads one line at a time
- readlines(): Reads all lines into a list
- write("text"): Writes the specified string
- writelines([list]): Writes a list of strings to the file
- seek(offset, whence): Moves file pointer to a position
- tell(): Returns the current position of the file pointer
- truncate(size): Resizes the file to the given number of bytes
- close(): Closes the file to free resources
- fileno(): Returns the file's descriptor used by the OS

seek() Parameters

- offset: Number of bytes to move the pointer
- whence: Reference point for movement
  - 0: From beginning (default)
  - 1: From current position
  - 2: From end of file (only in binary mode)

Binary Mode Examples

- "rb": Read binary file (like images)
- "wb": Write binary data
- "ab": Append to a binary file

Why close() Is Necessary

- Releases memory and system resources
- Flushes any data left in buffer to the file
- Ensures data is saved properly
- Helps avoid file corruption and access issues

Objective Type Questions

1. Which function is used to open a file in Python?
2. Which mode is used to write to a file, overwriting its contents?
3. Which mode allows adding data at the end of a file?
4. What is the default mode in `open()` function?
5. Which method reads a file line by line?
6. Which method is used to write data to a file?
7. Which method returns the current file pointer position?
8. Which method moves the file pointer to a specific location?
9. Which method resizes the file to a specific size?
10. Which statement is used to automatically close a file in Python?
11. What is the name of the method to get the file descriptor?
12. What is the mode to read and write in the same file, without truncating it?
13. Which method is used to close an open file?
14. What type of files are opened with mode `'rb'`?
15. Which keyword is used to handle files safely, ensuring they are closed automatically?

Answers to Objective Type Questions

1. open
2. w
3. a
4. r
5. readline
6. write
7. tell
8. seek
9. truncate
10. with
11. fileno
12. r+
13. close
14. binary
15. with

Assignments

1. **Explain different file opening modes in Python** with suitable examples for each. Illustrate how the behavior of the file changes when using modes like 'w', 'a', 'r', 'r+', and 'w+'.
2. **Write a Python program that performs the following operations**:
3. Creates a new file and writes three lines of text to it.
4. Appends two more lines to the same file.
5. Reads the entire content and displays it on the screen. Include appropriate comments and explain the purpose of each file mode used.

6. **Describe the purpose of the following file methods** in Python with syntax and examples:

7. read(), readline(), write(), seek(), tell(), truncate(), close() Also explain what happens if these methods are used incorrectly.

8. Discuss the significance of the with statement in Python file handling. Compare file operations done with and without with. Write sample programs to demonstrate both approaches and explain the difference in terms of resource management and error handling.

Reference

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

2. Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.

3. Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). Green Tea Press.

4. Beazley, D., & Jones, B. K. (2013). *Python Cookbook* (3rd ed.). O'Reilly Media.

Suggested Reading

1. Beazley, D., & Jones, B. K. (2013). *Python Cookbook* (3rd ed.). O'Reilly Media.

2. VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media.

Web Resources

3. Python Software Foundation. (n.d.). *The Python Tutorial*. https://docs.python.org/3/tutorial/

4. W3Schools. (n.d.). *Python Tutorial*. https://www.w3schools.com/python/

Unit 2

Advanced File Operations

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

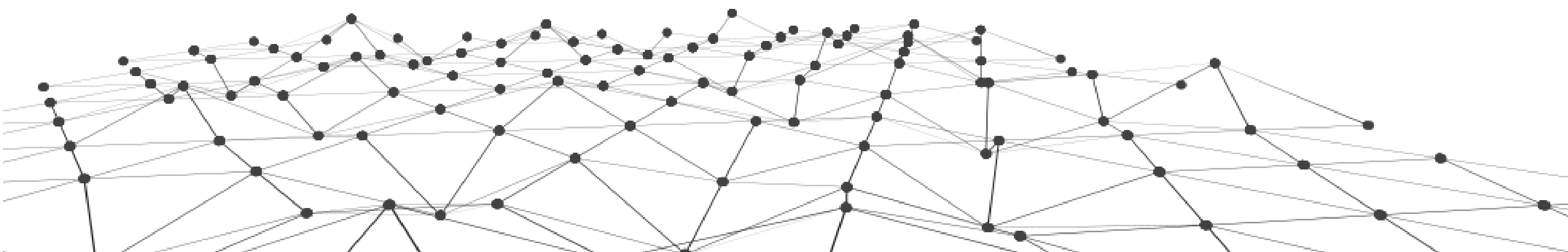
- Familiarize file handling operations such as opening, reading, writing, and closing files in Python.
- Demonstrate the use of file pointer methods like seek(), tell(), and truncate() to manipulate file data and cursor positions.
- Differentiate between text files and binary files, and perform appropriate operations on each.
- Implement safe file access using the with statement and understand the internal working of context managers.
- Use custom context manager classes __enter__() and __exit__() methods to handle file resources efficiently and prevent resource leaks.

Prerequisites

Imagine an online examination system that stores each student's answers and scores for future reference. In such a system, Python file operations like `open()`, `write()`, and `read()` are used to handle student data efficiently. For example, the program can use `write()` to save each student's responses into a text file and `readlines()` to fetch the stored answers for evaluation. The `seek()` and `tell()` methods help in navigating through large files to locate or update specific data without reloading the entire content. Additionally, `truncate()` may be used to clear or reset result files before storing new entries. The use of the `with` statement ensures that files are automatically closed after operations, preventing data loss and memory leaks. File operations also allow the system to access configuration files or question banks saved in `.txt` or `.csv` formats. These operations are critical for building reliable, data-driven Python applications where persistent storage and retrieval are necessary.

Key words

Append, mode, flush, close, exception, path, encoding



Discussion

4.2.1 File pointer methods

In Python, file pointer methods are essential tools that help manage how data is read from or written to a file. These methods allow the programmer to control the position of the file cursor, making file operations more flexible and efficient. For example, the seek() method moves the file pointer to a specific byte location, which is useful when you want to skip over or revisit certain parts of a file. The tell() method returns the current position of the file pointer, helping to keep track of where operations are taking place within the file. These methods are particularly valuable when working with large files, where reading the entire file at once would be inefficient. They also support both text and binary modes of operation, although some limitations exist based on the mode used. Proper use of file pointer methods ensures precise navigation within files and contributes to effective file handling in Python programs.

Python provides robust support for file operations and includes a range of built in functions that allow the creation, reading, writing, and manipulation of files. Python can handle two main types of files - text files that store data in human readable form and binary files that store data in a format readable by machines.

Text files: These files contain characters encoded using a standard character encoding scheme such as ASCII or UTF-8. Each line in a text file is terminated with a special character sequence called the End of Line. In Python, this is represented by the newline character '\n', which signals the end of a line and the beginning of the next. Text files are typically used for storing structured or unstructured plain text, such as logs, configuration files, or documents.

Binary files: Unlike text files, binary files store data as a continuous stream of bytes. These files do not include any line terminators. The data is encoded in binary format, which can represent complex data types such as images, audio, video, or serialized Python objects. When writing to or reading from binary files, Python reads the exact byte content using specific modes and functions. Handling binary files usually requires the use of the rb (read binary) or wb (write binary) modes to ensure that the byte stream is preserved accurately without any encoding or newline translation.

4.2.1.1 seek() function in Python

The seek() function in Python is used to change the position of the file cursor to a specific location within a file. This feature lets you read from or write to any part of the file, not just from the beginning. For instance, if you want to ignore the first 10 characters while reading a file, you can use the following code:

```
f = open("demo.txt", "r")

f.seek(10)

print(f.read())

f.close()
```

This code moves the file cursor to the tenth character and starts reading from that point.

Syntax:

file.seek(offset, from_what)

Parameters:

offset: The number of bytes to move the cursor.

from_what: (optional) The point of reference from where the cursor movement begins:

0 - sets the reference at the beginning of the file

1 - sets the reference at the current position in the file

2 - sets the reference at the end of the file

Returns:

The function returns the new absolute cursor position from the start of the file.

Note:

When working in text mode, only 0 can be used as the from_what value. If you want to use 1 or 2, the file must be opened in binary mode using ‘rb’.

Example 1:

Using seek() in Text Mode

Assume that the file Seek_example1.txt contains the following line:
I am a third semester BSc.DSA Student.

```
f = open("Seek_example1.txt", "r")

f.seek(22)

print(f.tell())

print(f.readline())

f.close()
```

Output:

```
22

BSc.DSA Student.
```

Explanation:

The seek(22) command moves the file cursor to the 22nd character.

The tell() function confirms that the cursor is currently at position 22.

The readline() function then reads the line starting from that position.

Example 2: Using seek() in Binary Mode with Negative Offset

Assume that the file seek_example2.txt contains the following binary content:

I am a third semester BSc. DSA Student.

```
f = open("seek_example2.txt", "rb")

f.seek(-11, 2)

print(f.tell())

print(f.readline().decode('utf-8'))

f.close()
```

Output:

```
47

DSA Student.
```

Explanation:

The file is opened in binary read mode ("rb").

The function seek(-11, 2) positions the cursor 11 bytes before the file's end.

readline() starts reading from that position until the end of the file.

The output is converted from binary format to a readable string using decode.

4.2.1.2 Python tell() function

tell() method

Access modes determine the kinds of operations that can be performed on an opened file. They specify how the file will be used after opening. These modes also decide the position of the file pointer, also known as the file handle. A file handle acts like a cursor that shows where data will be read from or written into the file. At times, it becomes necessary to check the current position of the file handle. The tell() method helps in finding out the current location of the file handle. This method returns the present position of the file object in the form of an integer value.

The tell() method does not take any arguments. When a file is opened, unless it is opened in append mode, the file pointer starts at the beginning of the file. Therefore, the initial value returned by tell() is zero.

Syntax:

```
file_object.tell()
```

Examples

Example 1: Using tell() right after opening a file

```
# Open a file in read mode
file = open("myfile.txt", "r")
# Get the current file pointer position
position = file.tell()
print("Current file pointer position:", position)
file.close()
```

Output:

```
Current file pointer position: 0
```

This is because the file pointer starts at the beginning when the file is opened in read mode.

Example 2: Using tell() after reading some characters

```
# Open a file in read mode
file = open("myfile.txt", "r")
# Read first 5 characters
data = file.read(5)
# Check file pointer position
position = file.tell()
print("After reading 5 characters, pointer is at position:", position)
file.close()
```

Output:

```
After reading 5 characters, pointer is at position: 5
```

Example 3: Using tell() in a file opened in write mode

```
# Open a file in write mode
file = open("sample.txt", "w")
# Write some text
file.write("Hello World")
# Check file pointer position
```

```
position = file.tell()

print("Pointer position after writing:", position)

file.close()
```

Output:

```
Pointer position after writing: 11
```

The pointer is at 11 because the string "Hello World" has 11 characters.

4.2.2 Python truncate() Method

The truncate() method in Python is used to change the size of a file. This method allows you to either reduce or increase the file's size, depending on the value passed to it.

Functionality

When you call truncate(), it cuts off the file at the specified size, discarding any data beyond that point. If you do not provide a size argument, the file will be truncated at the current file pointer position (i.e., wherever the cursor is currently located within the file). Importantly, this operation does not move the file pointer, its position remains unchanged after the method call.

Behavior Based on Size Argument

- If the specified size is less than the current size of the file, the extra data will be permanently removed.If the size is greater than the current file size, what happens depends on the operating system:
  - The file may stay unchanged.
  - The file might expand to the new size, with the extra bytes filled with zero (\x00) values.
  - Alternatively, the new content added may be undefined or garbage data, depending on the platform.

File Mode Requirement

To use truncate(), the file must be opened in a mode that allows writing—such as write ('w') mode, append ('a') mode, or read and write ('r+') mode. Attempting to truncate a file opened in read-only mode will raise an error.

Syntax

```
fileObject.truncate(size)
```

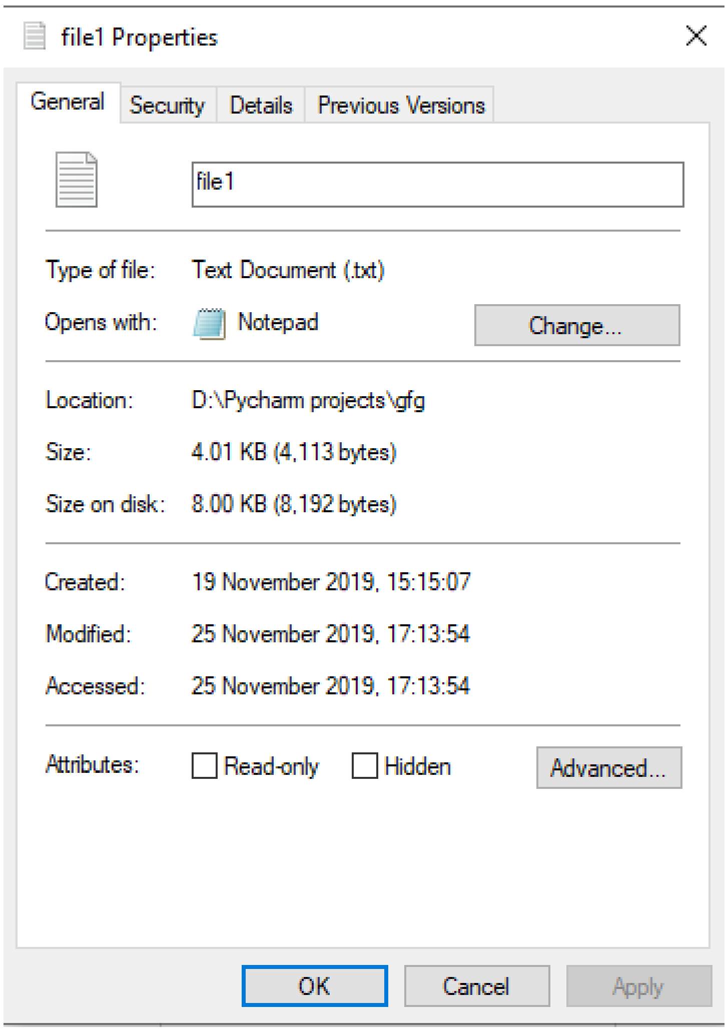
- fileObject: A file object returned by open()
- size (optional): An integer value representing the desired file size in bytes

Example

```
with open("sample.txt", "w+") as f:
    f.write("Hello, world!")
    f.truncate(5)
```

After this operation, the contents of sample.txt will be "Hello" , the rest is removed.

Example: See the below image for file size.



file1 Properties

General | Security | Details | Previous Versions

file1

Type of file: Text Document (.txt)
Opens with: Notepad

Location: D:\Pycharm projects\gfg
Size: 4.01 KB (4,113 bytes)
Size on disk: 8.00 KB (8,192 bytes)

Created: 19 November 2019, 15:15:07
Modified: 25 November 2019, 17:13:54
Accessed: 25 November 2019, 17:13:54

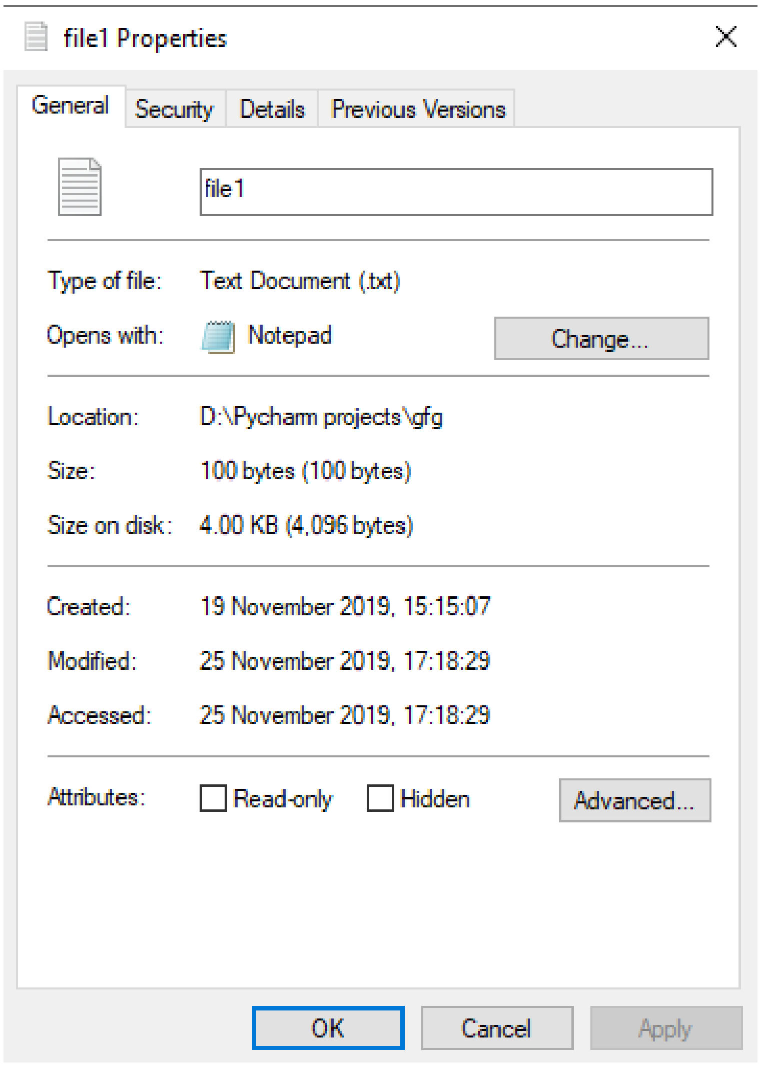
Attributes: ☐ Read-only ☐ Hidden

Let's change the file size to 100 bytes.

```
# Python program to demonstrate # truncate() method
fp = open('file1.txt', 'w')
# Truncates the file to specified # size
fp.truncate(100)
```

```
# Closing files
fp.close()
```

Output:



file1 Properties

General | Security | Details | Previous Versions

file1

Type of file: Text Document (.txt)

Opens with: Notepad

Location: D:\Pycharm projects\gfg

Size: 100 bytes (100 bytes)

Size on disk: 4.00 KB (4,096 bytes)

Created: 19 November 2019, 15:15:07

Modified: 25 November 2019, 17:18:29

Accessed: 25 November 2019, 17:18:29

Attributes: ☐ Read-only ☐ Hidden

The two examples demonstrate different uses of Python's truncate() method and how it affects file content and size. In the first example, the file sample.txt is opened in write-and-read mode ("w+"), and the string "Hello, world!" is written to it. Then, the truncate(5) method is called, which reduces the file size to 5 bytes. As a result, only the first five characters, "Hello", are retained in the file, and the rest of the content is permanently removed. This shows how truncate() can be used to shorten a file after data has been written. In contrast, the second example opens a new file file1.txt in write mode ("w") without writing any content. When truncate(100) is called, it expands the file size to 100 bytes, even though no actual text is added. The file will appear empty but will occupy 100 bytes, typically filled with null bytes (\x00) or undefined content, depending on the operating system. This demonstrates that truncate() can also be used to pre-allocate file space or modify a file's size without writing visible content. The key difference lies in their intent and effect, one modifies existing content, while the other changes the file's size in the absence of content.

4.2.3 Using the with Statement in File Handling

In earlier approaches, whenever a file is opened, it must be explicitly closed using the close() method. If this step is forgotten, it can lead to several problems, for example, changes made to the file might not be saved properly until the file is closed. To avoid such issues, Python provides the 'with' statement.

The 'with' statement is used to manage resources like files more efficiently and safely. It simplifies the code by automatically handling the opening and closing of files. This makes the code cleaner, easier to read, and reduces the chances of bugs. When using with, there is no need to explicitly call file.close(), the file is automatically closed once the block of code inside the 'with' statement finishes execution, even if an error occurs.

Program Execution with the 'with'

When the following program is run:

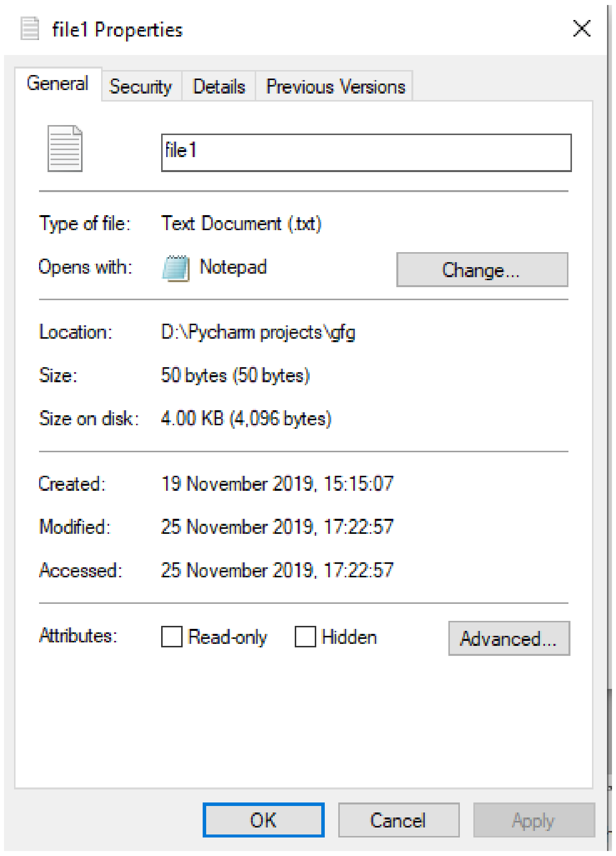
```
# Python program demonstrating

# truncate() method with the 'with' statement

with open('file1.txt', 'w') as fp:

fp.truncate(50)
```

Here's what happens step by step:

1. The file file1.txt is opened in write mode ('w') using the with statement.
2. Inside the block, the truncate(50) method is called, which resizes the file to 50 bytes.
   - ♦ If the file was previously larger than 50 bytes, it gets shortened.
   - ♦ If it was smaller, the file may be extended to 50 bytes with null bytes (\x00), depending on the platform.
3. Once the block is completed, Python automatically closes the file, ensuring all changes are properly saved and no file resources are left open.

This approach helps avoid common mistakes and makes the file-handling process more secure and efficient.

Output:


file1 Properties

General | Security | Details | Previous Versions

file1

Type of file: Text Document (.txt)
Opens with: Notepad
Location: D:\Pycharm projects\gfg
Size: 50 bytes (50 bytes)
Size on disk: 4.00 KB (4,096 bytes)
Created: 19 November 2019, 15:15:07
Modified: 25 November 2019, 17:22:57
Accessed: 25 November 2019, 17:22:57
Attributes: Read-only Hidden

4.2.4 Working with File Renaming and Deletion in Python

Python provides built-in support for renaming and deleting files through the operating system module. These operations are essential for effective file system management, especially when organizing, updating, or cleaning up files.

4.2.4.1 Renaming Files in Python

To rename a file, Python offers the os.rename() function, which allows you to change a file's name easily. This function requires two arguments: the current name of the file and the new name you want to assign.

Syntax:

os.rename(existing_filename, new_filename)

Parameters:

- existing_filename: The current name of the file you wish to rename.
- new_filename: The new name that will replace the existing one.

Example: Renaming a File

Let's say you want to rename a file named oldfile.txt to newfile.txt. Here's how you can do it using Python:

```
import os
# Original file name
current_name = "oldfile.txt"
# Desired new name
new_name = "newfile.txt"
# Renaming the file
os.rename(current_name, new_name)
print(f'File '{current_name}' renamed to '{new_name}' successfully.")
```

Output:

```
File 'oldfile.txt' renamed to 'newfile.txt' successfully.
```

4.2.4.2 Deleting Files in Python

In Python, files can be deleted using the os.remove() function from the os module. This function permanently deletes the file whose name is passed to it.

Syntax

```
os.remove(file_name)
```

Parameter

- file_name: The name (and optionally the path) of the file you want to delete. This should be passed as a string.

Example

The following example demonstrates how to delete a file named "file_to_delete.txt":

```
import os
# Name of the file to be removed
file_for_deletion = "myfilenew.txt"
# Remove the file
os.remove(file_for_deletion)
print(f'File '{file_for_deletion}' deleted successfully.")
```

Output

```
File 'myfilenew.txt' deleted successfully.
```

Explanation

When this script is executed:

1. The file “myfilenew.txt” is identified.
2. The os.remove() function deletes the file from the system.
3. A confirmation message is printed once the deletion is successful.

Make sure the file exists before calling os.remove(), or it will raise a FileNotFoundError.

4.2.5 Reading a File Line by Line in Python

Python offers built-in functions to create, write, and read files. It supports handling both text files and binary files (which store data in the form of 0s and 1s). In this section, we’ll focus on how to read the contents of a text file one line at a time.

Example:

```
with open(‘filename.txt’, ‘r’) as file:
    for line in file:
        print(line.strip())
```

Explanation:

- The open() function is used to open the file named “filename.txt” in read mode (‘r’).
- The with statement ensures that the file is properly closed after the operation is completed.
- The for loop reads the file one line at a time.
- line.strip() is used to remove any leading/trailing whitespace or newline characters before printing each line.

This approach is memory-efficient, especially useful when working with large files, as it reads and processes one line at a time instead of loading the entire file into memory.

4.2.5.1 Reading a File Using a Loop

When a file is opened using the open() function, it returns an iterable file object. One efficient way to read a file line-by-line is by using a for loop to iterate over this file object directly. This approach makes use of Python’s built-in capability to process the file line-by-line without needing to call any explicit method like readline() or readlines().

Example

```
# Creating a sample file with multiple lines

L = [“BSc.\n”, “Data Science\n”, “and Analytics\n”]

file1 = open(‘myfile.txt’, ‘w’)

file1.writelines(L)
```

```
file1.close()

# Opening the file in read mode

file1 = open('myfile.txt', 'r')

count = 0

print("Using for loop")

# Iterating over the file object line by line

for line in file1:

    count += 1

    print("Line{}: {}".format(count, line.strip()))

file1.close()
```

Output

```
Using for loop

Line1: BSc.

Line2: Data Science

Line3: and Analytics
```

Explanation

1. A list of strings is written to a file named "myfile.txt".
2. The file is reopened in read mode.
3. A for loop is used to go through each line in the file one at a time.
4. Each line is stripped of extra whitespace (like newline characters) using strip() and printed along with its line number.
5. Finally, the file is closed.

This method is memory-efficient and especially useful for reading large files, as it processes one line at a time rather than loading the entire file into memory.

4.2.5.2 Reading Files Using List Comprehension

List comprehension in Python is a concise way to create lists by placing an expression inside square brackets, along with a for loop to iterate over elements. When applied to file handling, it can be used to read lines from a file efficiently.

In this example, we demonstrate two ways of reading a file using list comprehension:

1. Reading with newline characters included

2. Reading with newline characters removed

Example

```
# Reading file lines with newline characters preserved
with open('myfile.txt') as f:
   l = [line for line in f]
print(l)
# Reading file lines with newline characters removed
with open('myfile.txt') as f:
   l = [line.rstrip() for line in f]
print(l)
```

Output

```
['BSc.\n', 'Data Science\n', 'and Analytics\n']
['BSc.', 'Data Science', 'and Analytics']
```

Explanation

- In the first with block, each line from the file is added to the list l as it is, including the \n newline characters.
- In the second block, the rstrip() method is used to strip off the trailing newline characters from each line before adding it to the list.
- The result is printed in both cases, showing the difference between raw file lines and cleaned lines.

This approach is compact and useful when you want to quickly read and process lines from a file into a list.

4.2.5.3 Reading Files Using readlines()

The readlines() function in Python reads all lines from a file at once and stores them as individual string elements in a list. Each element in this list represents a line from the file, including the newline characters (\n).

This method is suitable for small files since it loads the entire file content into memory. After reading the lines, we can loop through the list and use the strip() method to remove the newline characters from each line.

Example

```
# Creating a file and writing multiple lines
L = ["BSc.\n", "Data Science and\n", "Analytics\n"]
```

```
file1 = open('myfilenew.txt', 'w')
file1.writelines(L)
file1.close()
# Reading the file using readlines()
file1 = open('myfilenew.txt', 'r')
Lines = file1.readlines()
count = 0
# Iterating through the list of lines
for line in Lines:
    count += 1
    print("Line{}: {}".format(count, line.strip()))
```

Output

```
Line1: BSc.
Line2: Data Science and
Line3: Analytics
```

Explanation

- A list of strings is written to a file called "myfilenew.txt", each ending with a newline character.
- The file is then opened in read mode, and readlines() reads all lines into a list named Lines.
- A for loop goes through each line in the list, using strip() to remove extra whitespace or newline characters.
- Each cleaned line is printed along with its line number.

This method is simple and effective for reading files that are not too large to fit in memory.

4.2.6 Using the with Statement in Python

When working with files in Python, it's crucial to ensure that files are properly closed after operations. Forgetting to close a file can lead to issues like data not being saved or system resources being unnecessarily occupied. Typically, this is done using the file.close() method.

Python's with statement simplifies file handling by automatically managing the opening and closing of files. Once the code inside the with block finishes executing-even if an error occurs, the file is safely closed. This approach leads to cleaner, more reliable code with less risk of mistakes.

Example 1: Writing to and Reading from a File Using with Statement

```
# Writing content to the file using with statement
L = [“BSc.\n”, “Data Science\n”, “and Analytics\n”]
with open(“myfile.txt”, “w”) as fp:
    fp.writelines(L)
     # Reading the file using readlines()
count = 0
print(“Using readlines()”)
with open(“myfile.txt”) as fp:
    l = fp.readlines()
    for line in l:
        count += 1
        print(“Line{}: {}”.format(count, line.strip()))
# Reading the file using readline()
count = 0
print(“\nUsing readline()”)
with open(“myfile.txt”) as fp:
    while True:
        count += 1
        line = fp.readline()
        if not line:
            break
        print(“Line{}: {}”.format(count, line.strip()))
# Reading the file using a for loop
count = 0
print(“\nUsing for loop”)
with open(“myfile.txt”) as fp:
    for line in fp:
        count += 1
        print("Line{}: {}".format(count, line.strip()))
```

Output

```
Using readlines()
Line1: BSc.
Line2: Data Science
Line3: and Analytics
Using readline()
Line1: BSc.
Line2: Data Science
Line3: and Analytics
Using for loop
Line1: BSc.
Line2: Data Science
Line3: and Analytics
```

Explanation

- with open(...): ensures that the file is properly closed after the block completes.
- readlines() reads all lines at once and returns them as a list.
- readline() reads one line at a time in a loop.
- Using a for loop directly on the file object allows line-by-line reading in a more Pythonic way.

All three reading methods produce the same output, and using the with statement ensures that the file is always closed safely, regardless of which reading method is used.

Example 2: Without with (Manual Closing)

```
file = open("example.txt", "r")
try:
content = file.read()
print(content)
finally:
```

```
file.close()  # Ensures the file is closed
```

Output:

```
Hello, World!
```

Explanation:

This approach opens "example.txt" in read mode, reads and prints its content, and then manually ensures that the file is closed using a finally block.

Example 3: With with (Automatic Closing)

```
with open("example.txt", "r") as file:
content = file.read()
print(content)
```

Output:

```
Hello, World!
```

Explanation:

The with statement handles the opening and closing of the file automatically. Once the block completes, the file is closed without needing a finally block.

4.2.6.1 Advantages of the with Statement

- Automatic Resource Management: Ensures resources are acquired and released properly.
- No Need for Try-Finally: Replaces traditional try-finally blocks used for manual cleanup.
- Improved Readability: Reduces boilerplate code, making programs easier to read and maintain.

a. Common Use : File Handling

The with statement is most commonly used with the open() function for file operations.

Example: Reading a File

```
with open("example.txt", "r") as file:
contents = file.read()
print(contents)
```

Output:

```
Hello, World!
```

Explanation:

Opens “example.txt” in read mode and automatically closes it after reading.

Example: Writing to a File

```
with open(“example.txt”, “w”) as file:
file.write(“Hello, Python with statement!”)
```

Output:

```
Hello, Python with statement!
```

Explanation:

The file is opened in write mode and written to. Upon exiting the block, the file is automatically closed.

Comparison: with vs. Without with in Writing

Without with (Manual File Closure)

```
file = open(“example.txt”, “w”)
try:
file.write(“Hello, Python!”)
finally:
file.close()
```

Output:

```
Hello, Python!
```

Explanation:

The file is manually opened and closed using a try-finally block to avoid resource leaks.

With with (Automatic File Closure)

```
with open(“example.txt”, “w”) as file:
file.write(“Hello, Python!”)
```

Output:

```
Hello, Python!
```

Explanation:

The file is written and automatically closed by the with block, making the code cleaner and safer.

c) Understanding Context Managers in with

The with statement relies on context managers, which define how resources are set up and cleaned up. A context manager must define two methods:

- __enter__() – Acquires the resource and returns it.
- __exit__() – Releases the resource when exiting the block.

Example: Custom Context Manager for File Writing

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file
    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()
with FileManager('example.txt', 'w') as file:
        file.write('Hello, World!')
```

Output:

```
Hello, World!
```

Explanation:

- __init__() sets the filename and mode.
- __enter__() opens the file.
- __exit__() ensures it's closed after writing.
- This custom class behaves just like the built-in file context manager used with open().

Recap

- File pointer methods like seek() and tell() are used to control the cursor position within a file.
- The seek() method moves the file pointer to a specific byte offset, allowing random access to file contents.
- The tell() method returns the current file pointer position as an integer.
- In text mode, the seek() method only supports from_what=0 (beginning of file).
- For from_what=1 (current position) or 2 (end of file), the file must be opened in binary mode (e.g., 'rb').
- Using seek(offset) is helpful when skipping specific parts of a file or revisiting previous content.
- Binary files must be handled using modes like 'rb' or 'wb' to avoid newline translations.
- Text files contain characters encoded in schemes like UTF-8 and are human-readable.
- Binary files store data in bytes and are used for images, audio, video, and other non-text content.
- The truncate() method resizes a file to a specified number of bytes, either shrinking or expanding it.
- If truncate() is called with no size argument, it truncates the file at the current pointer position.
- When expanding a file using truncate(size), the new content may be filled with null bytes (\x00) or be undefined.
- truncate() can only be used if the file is opened in a mode that allows writing (e.g., 'w', 'a', 'r+').
- The with statement in Python automatically handles opening and closing files, even in case of errors.
- Using with makes file-handling code cleaner, safer, and less prone to bugs caused by forgetting file.close().
- Python provides built-in support for renaming and deleting files using os.rename() and os.remove().
- Reading a file line by line can be efficiently done using a for loop, saving memory when working with large files.

- The readlines() method reads all lines at once and returns them as a list; best for small files.
- File contents can also be read using list comprehension, optionally stripping newline characters.
- You can define a custom context manager using __enter__() and __exit__() to manage file resources similarly to the built-in open() function.

Objective Type Questions

1. Which method in Python moves the file cursor to a specific location?
2. Which method returns the current position of the file pointer?
3. What is the default value of `from_what` in `seek()` for text mode?
4. What value of `from_what` in `seek()` refers to the end of the file?
5. In which mode must a file be opened to use `from_what = 1 or 2` in `seek()`?
6. Which method is used to change the size of a file?
7. What character represents the end of a line in a Python text file?
8. What type of file stores data in a human-readable format?
9. What type of file stores data as a stream of bytes?
10. Which keyword ensures a file is automatically closed after operations?
11. Which Python module is used for file renaming and deletion?
12. Which function is used to rename a file in Python?
13. Which function is used to delete a file in Python?
14. What error is raised if you try to delete a non-existent file?
15. Which method reads all lines from a file into a list?
16. Which method reads one line at a time from a file?
17. Which loop is memory-efficient for reading large files line by line?
18. What does the `tell()` method return?
19. Which method in a custom context manager opens a resource?
20. Which method in a custom context manager closes the resource?

Answers to Objective Type Questions

1. seek()
2. tell()
3. 0
4. 2
5. rb
6. truncate()
7. \n
8. Text
9. Binary
10. with
11. os
12. rename()
13. remove()
14. FileNotFoundError
15. readlines()
16. readline()
17. for
18. position
19. enter()
20. exit()

Assignments

1. Explain the working of seek() and tell() methods in Python file handling. Describe the purpose of these file pointer methods with relevant syntax and examples. Include the significance of the from_what parameter in the seek() function and how its usage differs between text and binary modes.

2. Discuss the functionality of the truncate() method in Python. Explain how this method alters the file size and how it behaves when the specified size is smaller or larger than the current file size. Provide examples showing the use of truncate() in both shrinking and expanding files.

3. Compare and contrast text files and binary files in Python. Write about the differences in storage, structure, and access methods for both file types. Include examples of how each type is opened, read from, and written to in Python.

4. Describe the use of the with statement in Python file handling. Explain why the with statement is preferred over manually opening and closing files. Illustrate with examples showing how it enhances code safety and clarity. Also mention what happens behind the scenes using context manager methods like __enter__() and __exit__().

5. Write a Python program using a custom context manager to handle file operations. Create a class-based context manager using __enter__() and __exit__() methods that writes and reads data from a file. Explain how your program ensures safe and efficient file handling.

Unit 3

Basics of Object-Oriented Programming

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- define the term "class" in Python.
- list the key concepts of Object-Oriented Programming (OOP).
- identify the syntax used to create an object in Python.
- recall the different types of constructors in Python.
- familiarise the three main types of methods in a Python class

Prerequisites

Imagine you are building a software system to manage a pet shelter. The shelter has many dogs, each with different breeds, ages, and behaviors. You need a way to organize all this information clearly so that you can keep track of each dog's unique details and what they can do.

If you just use simple lists or separate variables, it becomes confusing and hard to manage as the number of dogs grows. How do you keep all their data and behaviors together? How can you reuse common features without rewriting code again and again?

This is where Object-Oriented Programming (OOP) in Python comes in. OOP helps you model real-world things like dogs as "objects" with their own data and actions. Using OOP, you can create classes that serve as blueprints for these objects, encapsulate data to keep it safe, reuse code through inheritance, and even let the same function work differently based on the context. By the end, you will be able to build structured, reusable, and maintainable programs that reflect real-world problems more naturally.

Key Concepts

Class, Object, Polymorphism, Encapsulation, Inheritance, Variables, Parametrized

Discussion

4.3.1 Introduction to OOPs

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It seeks to incorporate in programming real-world concepts like inheritance, polymorphism, encapsulation, etc. The fundamental idea behind OOPs is to combine the data and the functions that use it such that no other portion of the code may access it.

4.3.2 Key Concepts of OOPs

The **key concepts of Object-Oriented Programming in Python as in Fig 4.3.1** mean the fundamental principles that define how Python supports programming using objects and classes. These concepts help organize code around **objects** that combine data and behavior, making programs easier to understand, maintain, and reuse.

The main key concepts of OOP in Python are

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

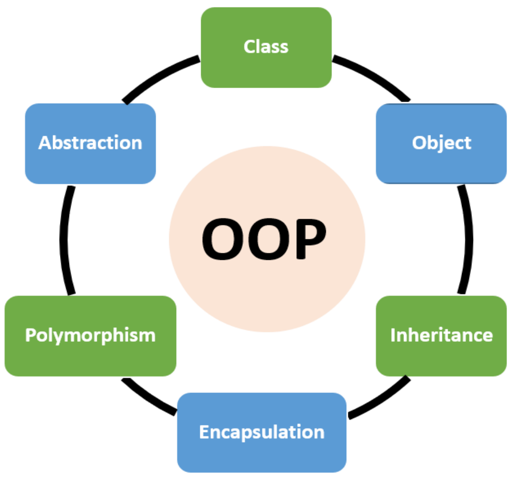


Fig 4.3.1 OOPs Concept

4.3.2.1 Class

Suppose you need to keep track of the number of dogs that might have various characteristics, such as breed or age. If a list is utilized, the dog's breed and age might be the first and second elements, respectively. What if there were 100 different breeds of dogs? How would you know which ingredient should go where? What if you wanted to give these dogs additional traits? This is unorganized and just what classes need.

A class serves as a blueprint for creating objects and represents a collection of those objects. It defines a set of attributes and methods that the objects (or instances) created from it will possess. In Python, classes are defined using the class keyword. Attributes, which are variables associated with a class, describe the properties of the objects. These attributes are publicly accessible and can be referenced using the dot (.) operator, for example, MyClass.MyAttribute.

Syntax: class ClassName:

```
  # Statement-1

  .

  .

  .

  # Statement-N
```

Example:

```
class Dog:                 #Define a class
   sound = "bark"          # Public class variable
   __type = "Canine"       # Private class variable
```

Using the class keyword, we built a class with the name dog in the example above.

- Public class members can be accessed from anywhere in the program, as all data members and functions are public by default unless stated otherwise.
- Protected members of a class, indicated by a single underscore (_), can only be accessed within the class itself and its subclasses.
- Private members of a class, marked with a double underscore (__), are accessible only within the class in which they are defined.

4.3.2.2 Objects

An **object** is a concrete instance of a **class**, containing its own specific data and functionality based on the class definition.

An object includes:

- **State**: Defined by its attributes, representing the characteristics or properties of the object.
- **Behavior**: Defined by its methods, showing how the object acts or responds to other objects.
- **Identity**: A unique identifier that distinguishes the object from others and allows interaction between different objects.

To better understand state, behavior, and identity, take the example of a Dog class. The identity of the dog can be its name, which makes it unique. The dog's state includes attributes like its breed, age, and color. Its behavior refers to actions it can perform, such as eating or sleeping. This helps show how an object (a dog) can have its own data and perform specific actions.

Creating an Object

In Python, creating an object means generating a new instance from a class. This process is known as Object Instantiation and involves using the class to construct an individual object with its own data and behavior.

Syntax: object_name = ClassName()

- ClassName is the name of the class already defined.
- object_name is the name of the object you are creating

Example:

```
class Dog:
    species = "Canine"              # Class attribute
    def __init__(self, name, age):
        self.name = name            # Instance attribute
        self.age = age              # Instance attribute
dog1 = Dog("Buddy", 3)              # Creating an object of the Dog class
print(dog1.name)                    #Accesses the instance attribute name of the dog1 object
print(dog1.species)                 #Accesses the class attribute species of the dog1 object
```

Output:

```
Buddy
Canine
```

- The self parameter refers to the current instance of the class. It is used to access that particular object's attributes and methods from within the class itself.
- The __init__ method serves as the constructor in Python and is automatically executed when a new object is instantiated. Its primary role is to initialize the class's attributes.

4.3.2.3 Polymorphism

Polymorphism is the concept of something existing in multiple forms. In simpler terms, it means that a single function or message can behave differently depending on the context. For example, one person may play various roles simultaneously like being a father, a husband, and an employee. Although it is the same person, their actions vary according to the role they are fulfilling. This flexibility in behavior based on different situations illustrates the idea of polymorphism. The different types of polymorphism is shown in Fig 4.3.2.

In object-oriented programming, polymorphism allows methods to perform different tasks based on the object calling them. It enhances code reusability and flexibility by allowing the same interface to be used for different data types. This concept is commonly implemented through method overriding and method overloading, making programs easier to scale and maintain.

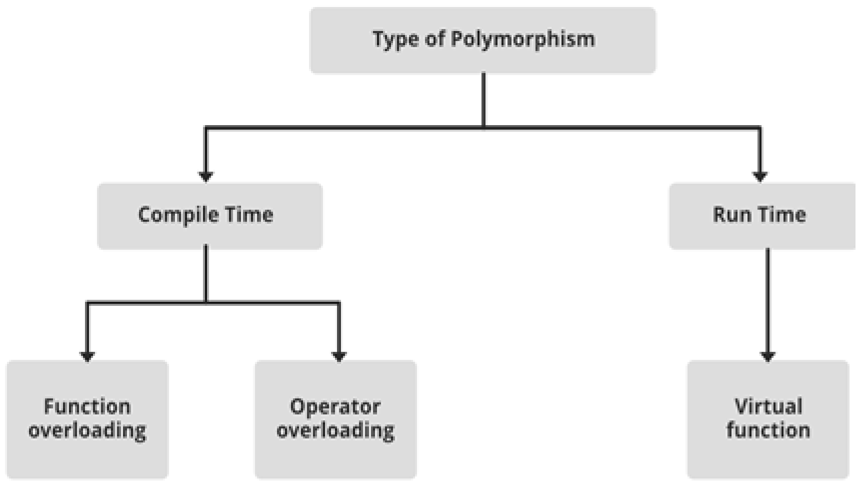


Fig 4.3.2 Types of Polymorphism

4.3.2.4 Encapsulation

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the concept of hiding the internal details of how an object works and only exposing a controlled interface to the outside world. In Python, this is achieved by wrapping data (variables) and methods (functions) into a single unit, usually a class, and restricting direct access to some of the object's components as shown in Fig 4.3.3. This protects the object's integrity by preventing unintended interference and misuse.

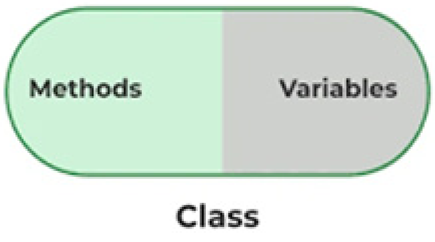


Fig 4.3.3 Encapsulation

Encapsulation allows programmers to define access levels for class members using public, protected, or private access modifiers. By doing so, sensitive data can be kept hidden from direct access and only modified through well-defined interfaces like getter and setter methods. This not only enhances data security but also makes the code more modular, maintainable, and easier to debug.

4.3.2.5 Inheritance

Inheritance is a key concept in object-oriented programming. It allows one class (called the child class or derived class) to inherit or receive the features (like variables and methods) of another class (called the parent class or base class). This means that the child class can use the code written in the parent class without rewriting it. Inheritance helps in code reusability, making programs shorter and easier to maintain.

If we have a class called Animal, we can create a child class Dog that inherits from Animal, meaning Dog will have all the features of Animal along with its own specific features. Inheritance can also be transitive, meaning if class B inherits from class A, and class C inherits from class B, then class C also inherits the features of class A.

Types of Inheritance

In Python, there are five main types of inheritance, each defining a different way in which classes relate to one another.

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

4.3.2.6 Data Abstraction

Data abstraction is a key and fundamental concept in object-oriented programming. It involves showing only the necessary details to the outside world while hiding the complex internal workings or implementation. For instance, consider a person driving a car. The driver knows that pressing the accelerator makes the car go faster and applying the brakes slows it down. However, the driver does not need to understand how the engine responds to these actions or the internal mechanics involved. This separation of functionality from the underlying process is what defines abstraction.

The main purpose of abstraction is

- To reduce complexity and make code easier to manage.
- To hide internal implementation from the user.
- To focus on what an object does and how it works.
- To provide a clear structure for creating reusable and extendable code.
- To improve security by hiding sensitive logic.

4.3.3 Python Variables

In Python, classes can contain two main types of variables

- Class Variables
- Instance Variables

1. Instance variables - variables that belong to each individual object created from a class, with every object having its own distinct copy. They are usually defined within methods, most commonly inside the __init__ method, using self.variable_name. The main purpose of instance variables is to store data or state that is unique and specific to each object.

```
class Dog:
    def __init__(self, name, breed):
        self.name = name             # Instance variable
        self.breed = breed           # Instance variable
  my_dog = Dog("Buddy", "Golden Retriever")
  your_dog = Dog("Lucy", "Labrador")
  print(my_dog.name)              # Output: Buddy
    print(your_dog.name)             # Output: Lucy
```

2. Class variables (Class Attributes)- variables that are shared among all instances of a class. Unlike instance variables, which belong to individual objects, class variables belong to the class itself and are defined directly within the class body, outside of any methods. They are used to store data that is common to every instance of the class or to hold constants related to the class, ensuring that this information is consistent and shared across all objects created from that class.

```
class Employee:
    raise_amount = 1.04                # Class variable
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)
  emp_1 = Employee("John", "Doe", 50000)
  emp_2 = Employee("Jane", "Smith", 60000)
  print(Employee.raise_amount)          # Output: 1.04
```

```
    print(emp_1.raise_amount)                # Output: 1.04 (accessed via instance)
```

4.3.4 Python Methods

In Python, a method is a function associated with an object that works with the object's data or carries out tasks related to it. Methods are defined inside a class and are accessed by calling them on an instance of that class using dot notation

Example: object.method().

Python includes three main categories of methods:

- ♦ Instance Methods
- ♦ Class Methods
- ♦ Static Methods

1. Instance Methods-These are the most frequently used methods in Python. They work on individual instances of a class. The first parameter, typically named self, refers to the particular object calling the method, enabling access to and modification of that object's attributes.

```
class MyClass:
  def __init__(self, value):
    self.value = value
  def get_value(self):          # Instance Method
    return self.value
obj = MyClass(10)
print(obj.get_value())
```

2. Class Methods-Class methods are tied to the class rather than any individual object. They are defined using the @classmethod decorator. The first parameter, usually named cls, refers to the class itself, giving these methods the ability to access and alter class-level attributes.

```
class MyClass:
  class_variable = “Hello”
  @classmethod
  def get_class_variable(cls):          # Class Method
    return cls.class_variable
print(MyClass.get_class_variable())
```

3. Static Methods-Static methods are much like regular functions but are placed inside a class to keep related functionality organized. They are defined using the @staticmethod decorator and do not take self or cls as their first argument. Since they do not rely on instance or class-specific data, static methods are ideal for utility tasks that are relevant to the class conceptually but do not require access to its internal state.

```
class MyClass:
    @staticmethod
    def static_greeting():          # Static Method
        return "This is a static greeting."
print(MyClass.static_greeting())
class MyClass:
    def __init__(self, value):
        self.value = value
    def get_value(self):        # Instance Method
        return self.value
obj = MyClass(10)
print(obj.get_value())
```

4.3.5 Constructors

In Python, constructors are special methods designed to initialize objects at the time of their creation. The main constructor method is __init__(), which is automatically executed when a new instance of a class is created. This method is typically used to set up instance variables or perform any setup tasks necessary for the object to operate as expected. Constructors are essential in object-oriented programming as they ensure that each object is initialized with the appropriate configuration when it is instantiated.

Syntax: def_init_(self):

# Body of the Constructor

4.3.5.1 Types of Constructors

In Python, there are three different kinds of constructors

- **Default Constructor**
- **Non-parameterized Constructor**
- **Parameterized Constructor**

1. Default Constructor

A default constructor in Python is the most basic form of constructor and does not take any parameters. If a class does not explicitly define a constructor, Python automatically provides one. This built-in default constructor is used to initialize the instance variables of a class with their default values. It is invoked automatically when an object of the class is created, ensuring that the object is properly set up even without custom initialization logic.

```
class Student:
def __init__(self):
self.name = “John Doe”
self.age = 18
self.grade = “A”
s = Student()
print(“Name:”, s.name)
print(“Age:”, s.age)
print(“Grade:”, s.grade)
```

Output:

```
Name: John Doe
Age: 18
Grade: A
```

In the given example, a default constructor is defined for the Student class. This constructor initializes the name, age, and grade attributes of the object with default values. When an object of the Student class is created using this constructor, it automatically assigns default values to these attributes.

2. Non-Parametrized Constructor

A non-parameterized constructor is a constructor defined by the programmer that does not take any arguments. Also known as a no-argument constructor, it is used to assign default values to the instance variables of a class when an object is created.

```
class Person:
def __init__(self):
self.name = “John”
self.age = 20
p = Person()
print(“Name:”, p.name)
print("Age:", p.age)
```

Output:

```
Name: John

Age: 20
```

In the given example, a non-parameterized constructor is defined for the Person class. This constructor sets the name attribute to "John" and the age attribute to 20 by default. When an object of the Person class is instantiated, these default values are automatically assigned to its name and age attributes.

3. Parametrized Constructor

A parameterized constructor in Python is a type of constructor that takes one or more arguments when an object is created. Unlike a default constructor, it allows the programmer to provide specific values during object instantiation, which are then used to initialize the instance variables of the class. This type of constructor is useful for setting unique values for each object, making the initialization process more flexible and customized.

```
class Rectangle:

def __init__(self, length, breadth):

self.length = length

self.breadth = breadth

r = Rectangle(10, 5)

print("Length:", r.length)

print("Breadth:", r.breadth)
```

Output:

```
Length: 10

Breadth: 5
```

In the given example, a parameterized constructor is defined for the Rectangle class. This constructor takes two parameters length and breadth and uses them to initialize the corresponding attributes of the object.

4.3.5.2 Constructor Overloading

Constructor overloading refers to the concept of having multiple constructors in a class, each with a different set of parameters. Although Python does not allow explicit constructor overloading as seen in some other languages, similar functionality can be accomplished by using default arguments and optional parameters within a single constructor.

Syntax: class MyClass:

def __init__(self, param1, param2=None):

# Constructor implementation

Parameters:

- self: Refers to the current instance of the class.
- param1: A required parameter used for initialization.
- param2: An optional parameter with a default value of None, which is used if no value is supplied.
- None: An optional parameter with a default value of None. If not provided, it takes on this default value.

```
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self.age = age
person1 = Person("Alice")
person2 = Person("Bob";25)
print(person1.name, person1.age)
print(person2.name, person2.age)
```

Output

```
Alice None
Bob 25
```

In this example, the Person class includes a constructor that takes name as a required parameter and age as an optional one. Two objects, person1 and person2, are created using different arguments. The output shows the name and age for each object, person1 has no age specified, so it displays None, while person2 has an age of 25.

Recap

- ♦ Python's Object-Oriented Programming (OOP) uses objects and classes to model real-world concepts like inheritance, polymorphism, and encapsulation.
- ♦ OOP combines data and functions to restrict access and improve code organization.
- ♦ Key OOP concepts: Class, Objects, Polymorphism, Encapsulation, Inheritance, and Data Abstraction.
- ♦ A **class** is a blueprint for objects; it defines attributes and methods.
- ♦ Attributes can be public, protected (single underscore _), or private (double underscore __).
- ♦ **Objects** are instances of classes, with unique state (attributes), behavior (methods), and identity.
- ♦ The `self` parameter inside methods refers to the current object instance.
- ♦ The __`init`__ method acts as a constructor to initialize objects.
- ♦ **Polymorphism** allows the same method or function to behave differently depending on the object context.
- ♦ **Encapsulation** hides internal object details and exposes controlled interfaces; access is controlled using public, protected, and private modifiers.
- ♦ **Inheritance** allows a child class to inherit properties and methods from a parent class, supporting code reuse.
- ♦ **Data Abstraction** hides complex internal implementation details and shows only essential features.
- ♦ Classes have two types of variables:
  - **Instance variables** are unique to each object.
  - **Class variables** are shared across all instances.
- ♦ Python methods include:
  - **Instance methods** (access individual object data via `self`).
  - **Class methods** (access class-level data via `cls`, marked with `@classmethod`).
  - **Static methods** (utility methods that do not access instance or class data, marked with `@staticmethod`).

- Constructors in Python:
  - **Default constructor**: automatically provided if none defined.
  - **Non-parameterized constructor**: sets default values without parameters.
  - **Parameterized constructor**: accepts parameters to initialize object attributes.

Objective Type Questions

1. What keyword is used to define a class in Python?
2. What is an instance of a class called?
3. What type of method does not take self or cls as a parameter?
4. Which OOP concept allows one class to inherit from another?
5. What is the term for hiding internal details in OOP?
6. What is the name of the concept where one function behaves differently based on context?
7. What OOP concept focuses on exposing only necessary details?
8. What is the name of the method used to initialize an object?
9. What type of constructor takes no parameters but is defined by the programmer?
10. What type of constructor automatically initializes with default values if none is defined?

Answers to Objective Type Questions

1. Class
2. Object
3. Static method
4. Inheritance
5. Encapsulation
6. Polymorphism
7. Abstraction
8. init
9. Non-parameterized constructor
10. Default constructor

Assignments

1. Explain the key concepts of Object-Oriented Programming (OOP) in Python. Illustrate each concept with a simple example.
2. Describe the difference between class variables and instance variables in Python. Provide code snippets to support your explanation.
3. Write a Python class called `Employee` with a parameterized constructor that initializes the employee's name, ID, and salary. Then, write a method to apply a raise to the salary based on a percentage. Create at least two employee objects and demonstrate the use of this method.
4. Discuss the role of constructors in Python with appropriate examples.
5. Write a Python program to define a class `Student` with private and public variables. Create an object and access both variables appropriately.

Reference

1. Zelle, J. (2010). *Python Programming: An Introduction to Computer Science* (2nd ed.). Franklin, Beedle & Associates.

2. Barry, P. (2016). *Head First Python* (2nd ed.). O'Reilly Media.

3. Subramaniam, V. (2021). *Programming Python with Object-Oriented Programming*. Pragmatic Bookshelf.

4. Beazley, D., & Jones, B. K. (2013). *Python Cookbook* (3rd ed.). O'Reilly Media.

5. Sharma, Y. (2020). *Object-Oriented Programming with Python: Learn OOP, design patterns, and testing*. Packt Publishing.

Suggested Reading

1. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

2. Guttag, J. V. (2016). *Introduction to Computation and Programming Using Python*. MIT Press.

3. Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.

4. Sweigart, A. (2019). *Automate the Boring Stuff with Python* (2nd ed.). No Starch Press.

5. https://docs.python.org/3/

Unit 4

Core Concepts of OOP in Python

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- familiarize the concept of classes and objects in Python.
- identify different types of inheritance in Python.
- explain polymorphism with examples.
- describe abstraction and encapsulation.
- recognize real-world uses of OOP in Python.

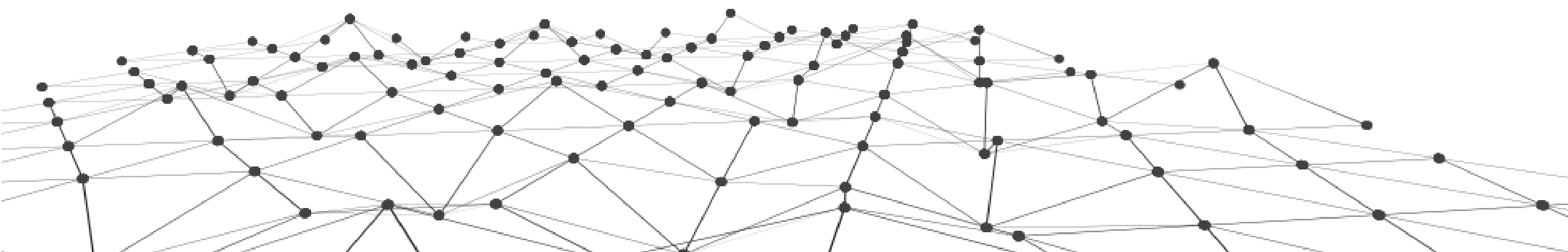
Prerequisites

Before learning Object-Oriented Programming (OOP) in Python, students should have a basic understanding of core Python concepts such as variables, data types, functions, conditional statements, and loops. These foundational topics help learners understand how data is stored and manipulated, and how logic is built in Python. Knowing how functions work, for example, lays the groundwork for understanding methods within classes. Without this foundation, diving into OOP may feel confusing or overwhelming.

We study OOP because it helps us model real-world systems more effectively in software. Just like in daily life, where we interact with objects such as a mobile phone (which has properties like color, brand, and behaviors like call or text), OOP lets us create similar structures in code. For example, if you're designing a school management system, a "Student" can be treated as an object with attributes like name and marks, and behaviors like enroll() or study(). This makes programs easier to build, understand, and maintain especially as they grow in size and complexity.

Key words

Class, Object, Inheritance, Polymorphism, Encapsulation



Discussion

In the real world, we interact with various objects every day like a car, a mobile phone, or a student. Each of these objects has properties (color, model, name, etc.) and behaviors (drive, ring, study, etc.).

Object-Oriented Programming (OOPs) is a method of structuring a program by bundling related properties and behaviors into individual objects. It is inspired by real-world systems where everything is treated as an object with characteristics (data) and actions (functions). Python is a powerful object-oriented programming language that allows developers to write clean, reusable, and organized code using the principles of OOP.

In OOP, a class serves as a blueprint for creating objects, and an object is an actual instance of a class containing real values. OOP helps break down complex problems into smaller, more manageable pieces by modeling them as interacting objects. This makes the program more modular and easier to maintain.

Python supports all major OOP principles like encapsulation, inheritance, polymorphism, and abstraction which help protect data, reuse code, provide flexibility, and hide complexity. Because of these features, OOP is widely used in developing software applications, from small scripts to large systems.

4.4.1 Inheritance

Inheritance is one of the core features of Object-Oriented Programming (OOP), which allows a class (called the **child class**) to inherit attributes and methods from another class (called the **parent class**). This makes programming more efficient, as it allows code reuse.

Python inheritance allows us to create a new class that uses the features of an existing class. It helps us reuse code and build relationships between classes.

Syntax

```
class Parent:

    # parent class code

class Child(Parent):

    # child class code
```

4.4.1.1 Types of Inheritance in Python

Python offers various types of inheritance based on the number of child and parent classes involved in the inheritance.

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance

4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance

Single inheritance in Python means one class inherits from one parent class. We use it when we want a child class to reuse or extend the functionality of a single base class.

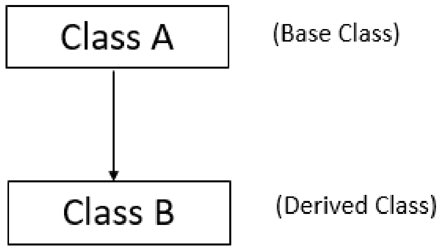


Fig 4.4.1 Single Inheritance

For example, if we define a class Vehicle with a method start(), we don't need to rewrite this method in a class Car; we can simply make Car inherit from Vehicle. In Python, this is done using parentheses like class Car(Vehicle):.

```
class Vehicle:
    def start(self):
        print("Vehicle started")
class Car(Vehicle):
    pass
c = Car()
c.start()  # Output: Vehicle started
```

This example demonstrates **single inheritance**, where a child class inherits from just one parent class. The class Car doesn't define its own start() method but can still use it because it's inherited from Vehicle.

2. Multiple Inheritance in Python

In multiple inheritance, a single subclass inherits from multiple superclasses. So, the child class can access attributes and methods from two or more parent classes.

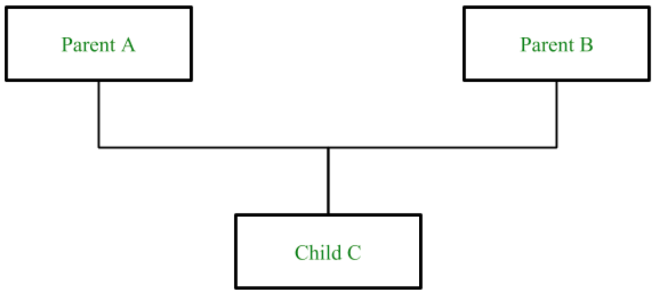


Fig 4.4.2 Multiple Inheritance

However, if the parent classes have methods with the same name, the child class uses the method from the first parent in the order they are listed.

For instance, imagine a class Father with a method work(), and another class Mother with a method care(). A class Child can inherit from both, gaining the ability to use both methods.

```
class Father:
    def work(self):
        print("Father works")
class Mother:
    def care(self):
        print("Mother cares")
class Child(Father, Mother):
    pass
c = Child()
c.work()   # Output: Father works
c.care()   # Output: Mother cares
```

3. Multilevel Inheritance

Python multilevel inheritance means a class inherits from a child class, which itself inherits from another parent class. This forms a chain of inheritance, passing features from one level to the next.

With multilevel inheritance in Python, we can build step-by-step class hierarchies, allowing each level to extend or reuse the previous one.

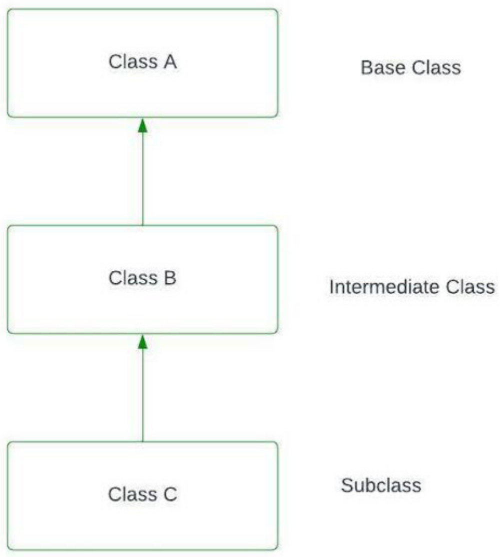


Fig 4.4.3 Multilevel Inheritance

A child class inherits from a parent, which in turn inherits from a grandparent. For example, class Grandparent has a method guide(), class Parent inherits from it and adds teach(), and class Child inherits from Parent. The Child class can now access both guide() and teach() methods.

```
class Grandparent:
    def guide(self):
        print("Grandparent guides")
class Parent(Grandparent):
    def teach(self):
        print("Parent teaches")
class Child(Parent):
    def learn(self):
        print("Child learns")
c = Child()
c.guide()   # Output: Grandparent guides
c.teach()   # Output: Parent teaches
c.learn()   # Output: Child learns
```

4. Hierarchical Inheritance

Hierarchical inheritance is the opposite of multiple inheritance. Therefore, more than one child class can be derived from a single-parent class.

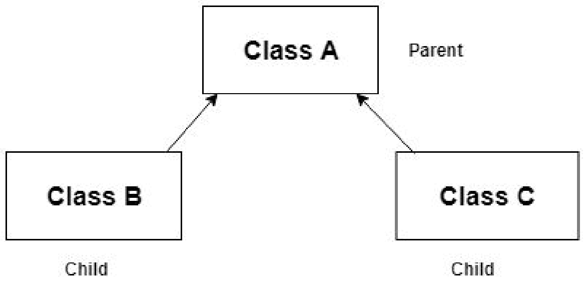


Fig 4.4.4 Hierarchical Inheritance

Hierarchical inheritance occurs when multiple child classes inherit from the same parent class. Suppose Dog and Cat both inherit from a class Animal that has a method eat(). Both subclasses automatically gain access to the eat() method.

```
class Animal:
    def eat(self):
        print("Animal eats")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
class Cat(Animal):
    def meow(self):
        print("Cat meows")
d = Dog()
c = Cat()
d.eat()   # Output: Animal eats
c.eat()   # Output: Animal eats
```

6. Hybrid Inheritance

Hybrid inheritance combines two or more types of inheritance. Hence, we can see multiple relationships between parent and child classes across different levels. We can say that hybrid inheritance in Python is a mixture of more than one inheritance style, like single, multiple, or multilevel.

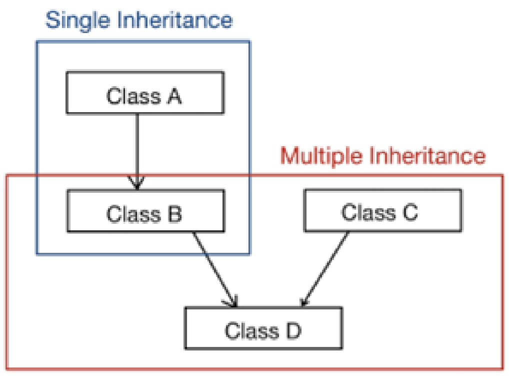


Fig 4.4.6 Hybrid Inheritance

4.4.1.2 Access Specifiers in Different Types of Inheritance

In Python, access specifiers (or access modifiers) control how members of a class (variables or methods) are accessed in inheritance. There are three types:

Public (x): Can be accessed in all child classes regardless of inheritance type. Also accessible from outside the class. Inherited and fully accessible by child classes.

Protected (_x): Meant to be accessed inside the class and its subclasses only. Inherited and accessible in all types of inheritance, but not intended for outside use. Inherited by child classes and accessible within them, but not recommended to access from outside.

Private (__x): Not inherited directly. Not accessible in child classes. Python internally name-manages private variables (e.g., _ClassName__x), so they are effectively hidden from Inheritance

Table 4.4.1 Accessibility of different classes

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

4.4.2 Polymorphism

Polymorphism in Python refers to the ability of a single function or method to operate differently based on the object it is used with. The term polymorphism originates from Greek, meaning many forms. In Python, this allows a method or function to adapt its behavior depending on the type of object invoking it.

For example, consider different animal classes where each animal produces a distinct sound. You can define a method named make_sound() that is used across all animal types, but each animal will produce its own specific sound. This means that the same method name can perform different actions depending on the object calling it.

```
class Dog:
def make_sound(self):
print("Bark")
class Cat:
def make_sound(self):
print("Meow")
dog = Dog()
cat = Cat()
dog.make_sound()
cat.make_sound()
```

#Output

```
Bark
Meow
```

Need of Polymorphism

- Ensures consistent interfaces across different classes.
- Allows objects to respond differently to the same method call.
- Promotes loose coupling by relying on shared behavior, not specific types.
- Enables writing flexible, reusable code that works across types.
- Simplifies testing and future extension of code.

4.4.2.1 Types of Polymorphism

Polymorphism in object-oriented programming (OOP) can be classified into two main types

1. Compile-time Polymorphism
2. Runtime Polymorphism

1. Compile-time Polymorphism (Static Polymorphism)

Compile-time polymorphism happens when the method to be called is determined at compile time, before the program is run. This type of polymorphism is achieved using method overloading or operator overloading.

Method overloading is when multiple methods have the same name but differ in the number or type of their parameters. The correct method is chosen based on the arguments passed when calling the method.

```
class MathOperations:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c
math = MathOperations()
print(math.add(5, 10, 15))
```

#Output

```
30
```

2. Runtime Polymorphism (Dynamic Polymorphism)

Runtime polymorphism happens when the method to be called is determined at runtime, during the execution of the program. This type of polymorphism is achieved using method overriding.

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in the parent class. The version of the method that is called depends on the object that is used to invoke it.

```
class MathOperations:

    def calculate(self, a, b):

        return a + b

class AdvancedMath(MathOperations):

    def calculate(self, a, b):

        return a * b

basic = MathOperations()

advanced = AdvancedMath()

print(&quot;Basic addition:&quot;, basic.calculate(5, 3))

print(&quot;Advanced multiplication:&quot;, advanced.calculate(5, 3))
```

#Output:

```
8

15
```

4.4.3 Abstraction and Encapsulation

Object-Oriented Programming (OOP) is a programming paradigm that helps organize code using objects. Two of its most essential concepts are Abstraction and Encapsulation. These concepts allow developers to build systems that are easier to understand, maintain, and scale.

Abstraction means hiding the complex details and showing only the essential features of an object or system. It focuses on what an object does, rather than how it does it. Encapsulation means binding the data (variables) and the code (methods) that manipulate the data into a single unit called class. It also helps protect data from being directly accessed or modified.

4.4.3.1 Abstraction

Abstraction is the concept of hiding the internal implementation details and showing only the essential features of the object.

Purpose

- Focuses on what an object does rather than how it does it.
- Helps in reducing complexity by suppressing lower-level details.

When you drive a car, you only need to know how to operate the steering wheel, accelerator, and brakes (what to do), not how the engine or transmission system works (how it's done). Achieved using abstract classes and methods from the abc module (Abstract Base Class).

```
from abc import ABC, abstractmethod

class Animal(ABC):

    @abstractmethod

    def sound(self):

        pass

class Dog(Animal):

    def sound(self):

        return "Bark"

d = Dog()

print(d.sound())

# Output: Bark
```

4.4.3.2 Encapsulation

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the concept of hiding the internal details of how an object works and only exposing a controlled interface to the outside world. In Python, this is achieved by wrapping data (variables) and methods (functions) into a single unit, usually a class, and restricting direct access to some of the object's components. This protects the object's integrity by preventing unintended interference and misuse.

Encapsulation allows programmers to define access levels for class members using public, protected, or private access modifiers. By doing so, sensitive data can be kept hidden from direct access and only modified through well-defined interfaces like getter and setter methods. This not only enhances data security but also makes the code more modular, maintainable, and easier to debug.

1. Public Members

Public members are class attributes (variables) or methods (functions) that can be accessed from anywhere, both inside and outside the class. In Python, by default, all members of a class are public unless explicitly specified otherwise.

```
class Student:

def __init__(self, name, age):
```

```
self.name = name

self.age = age

s = Student(&quot;Helen&quot;, 23) # Create an instance of Student

print(&quot;Name:&quot;, s.name)

print(&quot;Age:&quot;, s.age)
```

#Output

```
Name: Helen

Age: 23
```

In the above example, name and age are public members of the Student class. They can be accessed and modified directly using the objects.

2. Protected Members

Protected members are variables or methods that can be accessed within the class and its subclasses, but should not be accessed directly from outside the class. In Python, protected members are defined by prefixing the name with a single underscore

(e.g., _name).

```
class Student:

    def __init__(self, name, age):

        self._name = name

        self._age = age

s = Student(&quot;Bob&quot;, 21)

print(&quot;Name:&quot;, s._name)

print(&quot;Age:&quot;, s._age)
```

Output:

```
Name: Bob

Age:21
```

3. Private Members

Private members are variables or methods in a class that cannot be accessed directly from outside the class. In Python, we make something private by putting two underscores (__) in front of its name.

```
class Student:
    def __init__(self, name, marks):
        self.__name = name
        self.__marks = marks
    def display(self):
        print(&quot;Name:&quot;, self.__name)
        print(&quot;Marks:&quot;, self.__marks)
s = Student(&quot;Helen&quot;, 28)
s.display()
```

#Output:

```
Name: Alice
Marks: 28
```

4.4.4 Applications of OOP in Python

Object-Oriented Programming (OOP) in Python is widely used across various domains due to its ability to model real-world entities using objects and classes. Below are major areas where OOP is applied in Python.

1. GUI Applications (Graphical User Interfaces)

In GUI-based software, components like windows, buttons, forms, and dialog boxes are treated as objects. OOP helps organize these components into reusable classes. Each visual element has properties (such as size and color) and behavior (like click or drag). Python GUI libraries like Tkinter, PyQt, and Kivy use OOP to manage and manipulate interface elements effectively.

2. Game Development

Games involve multiple elements such as players, enemies, weapons, and levels, each of which can be modeled as objects. OOP allows developers to define shared behaviors using inheritance and override them where needed using polymorphism. Game development frameworks like Pygame leverage OOP to build scalable, interactive game environments with complex logic.

3. Web Development

In web development frameworks like Django and Flask, OOP plays a central role in managing views, templates, and databases. In Django, each database table is represented as a class, and each record as an object. This allows for better abstraction of backend logic, encourages modular coding practices, and simplifies the development and maintenance of large-scale web applications.

4. Data Science and Machine Learning

In data science, libraries like Pandas, NumPy, and Scikit-learn heavily utilize OOP. Data structures like data frames and models (e.g., regression, clustering) are implemented as classes. Objects can store data and provide methods to manipulate and analyze it. OOP facilitates clean, reusable workflows and simplifies experimentation with machine learning algorithms.

5. Simulation and Modelling

Simulations (e.g., financial systems, climate models, transportation systems) often mimic real-world behavior. OOP allows developers to represent each entity like a car, person, or account as an object. This approach provides clarity and flexibility in designing simulations where each object can interact with others and change state over time.

6. Enterprise Software Development

Enterprise software such as inventory systems, HR management tools, and billing systems require complex logic and data management. OOP enables the development of these systems in a modular way. Each business unit or functionality (like employee management, payroll, or product tracking) can be designed as a class, promoting separation of concerns and scalability.

7. Robotics and IoT (Internet of Things)

In robotics and IoT systems, hardware components like sensors, actuators, and motors are treated as objects. Each device has specific attributes and functions that can be encapsulated into a class. OOP helps in designing responsive systems where objects communicate with one another, process real-time data, and perform tasks like sensing, decision-making, and control.

8. Mobile Application Development

Python frameworks such as Kivy allow for mobile application development where screens, buttons, images, and events are modeled as objects. OOP helps define behavior and appearance consistently across different parts of the app. It supports an organized structure where features can be updated or extended without affecting the whole application.

9. API Development

APIs (Application Programming Interfaces), especially in RESTful web services, benefit from OOP as endpoints, requests, and responses can be modeled as classes. Business logic is encapsulated in objects, making it easier to manage routing, error handling, and data validation. Frameworks like FastAPI and Flask encourage object-based routing and service design.

10. Automation and Scripting

Even in automation tasks like file processing, email automation, or data extraction—OOP is beneficial. Scripts can be structured using classes that represent files, logs, reports, etc. This makes the script more flexible, reusable, and easier to maintain, especially as the complexity of tasks increases.

Recap

- OOP models real-world entities using classes (blueprints) and objects (instances).
- Inheritance allows child classes to reuse code from parent classes, supporting code efficiency.
- Polymorphism enables the same method name to behave differently across different classes.
- Abstraction hides implementation details, exposing only necessary functionality.
- Encapsulation protects data by restricting access to internal class details.
- Access specifiers (public, _protected, __private) define how class members are accessed.
- OOP is widely applied in fields like GUI development, games, web applications, ML, and automation.

Objective Type Questions

1. What is the blueprint for creating objects in Python OOP?
2. What do you call an instance of a class?
3. Which OOP principle allows reuse of code from a base class?
4. What is the OOP concept that allows one interface to have many implementations?
5. What term describes restricting access to parts of an object?
6. What hides internal implementation details from the user?
7. Which keyword is used to define a class in Python?
8. What symbol is used for protected members in Python?
9. What symbol is used to make a member private?
10. Which module is used for abstraction in Python?
11. Which type of polymorphism uses method overriding?

12. What type of inheritance involves one class inheriting from multiple classes?
13. Which inheritance type involves a class deriving from a class that already inherited another?
14. What is the default access specifier for class members in Python?
15. Which principle ensures that sensitive data is not directly accessible?

Answers to Objective Type Questions

1. Class
2. Object
3. Inheritance
4. Polymorphism
5. Encapsulation
6. Abstraction
7. class
8. Underscore (_)
9. Double underscore (__)
10. abc
11. Runtime
12. Multiple
13. Multilevel
14. Public
15. Encapsulation

Assignments

1. Define class, object, and constructor in Python with suitable examples.
2. Explain the five types of inheritance in Python with code examples.
3. Differentiate between compile-time and runtime polymorphism with appropriate Python examples.
4. Describe abstraction and encapsulation in OOP. How are they implemented in Python?
5. Discuss five real-world applications of OOP and explain how Python supports OOP in those domains.

Reference

1. Downey, A. B. (2015). Think Python: How to Think Like a Computer Scientist (2nd ed.). Green Tea Press.
2. Lutz, M. (2013). Learning Python (5th ed.). O'Reilly Media.
3. Martelli, A., Ravenscroft, A., & Ascher, D. (2005). Python Cookbook (2nd ed.). O'Reilly Media.

Suggested Reading

1. Sweigart, A. (2020). Automate the Boring Stuff with Python: Practical Programming for Total Beginners (2nd ed.). No Starch Press.
2. Hetland, M. L. (2017). Beginning Python: From Novice to Professional (3rd ed.). Apress.

5

BLOCK

Exception Handling and Database Programming

Unit 1
Exception Handling

Learning Outcomes

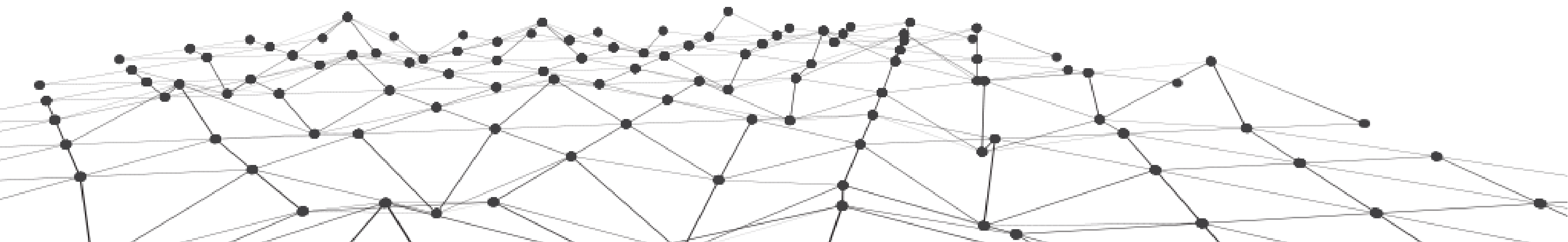
After the successful completion of the unit, the learner will be able to:

- to understand the concept of errors and exceptions in Python.
- to differentiate between various types of errors.
- to use Python's built-in exception handling mechanisms.
- to write robust programs that gracefully handle run-time errors.
- to create and raise custom exceptions.

Prerequisites

Imagine you are using an online ticket booking system. You have entered your details and selected your seats. When you click on the payment button, by mistake type your card number as a string of letters instead of numbers. Suddenly, the entire website crashes and shows a confusing error message. You are left wondering if your money was deducted or if your seats are gone. This happens when programmers fail to handle exceptions in their code. In real life, situations like this can damage a company's reputation and lead to loss of user trust. Exception handling in Python helps avoid such issues. It provides a structured way to manage errors that happen during the execution of a program. It ensures that programs do not crash unexpectedly and can respond gracefully to problems like invalid input, missing files, or network failures.

Exception handling is an essential skill for anyone learning Python because it strengthens a program's reliability and user experience. It trains learners to write robust code that can anticipate errors and deal with them effectively. Before studying this topic, learners should have a good understanding of Python basics such as variables, loops, conditionals, functions, and file operations. This topic helps in problem solving by encouraging careful thinking about what could go wrong in a program and how to respond. It improves code design by separating normal logic from error handling logic. The benefits include better program stability, cleaner debugging, easier maintenance, and a smoother experience for users. Studying exception handling makes learners more confident developers who can build real world applications that are smart, secure, and professional.



```
      +---------------------+
      |    try block        |
      +----------+----------+
                 |
      +----------v----------+
      |   Error occurs?     |
      +----+---------+------+
           |         |
       Yes |         | No
           |         |
  +----------v--+  +--v------------+
  | except block |  | else block   |
  +-------------+  +---------------+
        |
+--------v---------+
|  finally block   |
+------------------+
```

Key concepts

Exception Handling, Syntax Errors, Runtime Errors, Try and Except Blocks, Finally Block, Raise, Assert.

Discussion

5.1.1 Introduction to Exception Handling

Errors in a program can be broadly categorized into five types: **Compile Time Errors**, which occur during the compilation of the program and prevent it from executing; **Run Time Errors**, which arise while the program is running, such as division by zero, using an undefined variable, or trying to access a non-existent file; **Logical Errors**, which are mistakes in the program's logic that do not stop the execution but lead to incorrect results, like using the wrong operator precedence or an incorrect variable name in a calculation; **Syntax Errors**, which are issues in the structure of the code that stop it from running, including incorrect indentation, misspelled keywords, or missing symbols like colons, parentheses, or commas; and **Semantic Errors**, which are logic-related mistakes that lead to unintended or unexpected outcomes despite the code running without interruption.

Although a statement or expression may be syntactically valid, it can still lead to an error during program execution. For example, errors like attempting to open a non-existent file or dividing a number by zero may occur. These types of errors interrupt the normal flow of the program and are known as **exceptions**. In Python, an exception is an object that signifies the occurrence of an error. When such an error takes place during execution, an exception is said to be **raised**. It is the responsibility of the programmer to handle these exceptions to prevent the program from crashing unexpectedly. Therefore, while writing a program, a programmer should foresee potential issues that may arise and include appropriate code to manage such exceptions effectively.

Imagine you're creating a banking application where users enter the amount they wish to withdraw from their account. Now, suppose a user accidentally types text instead of a number like entering "two thousand" instead of 2000. The program will crash with an error, confusing the user and potentially losing unsaved data.

Example:

```
amount = int(input(“Enter withdrawal amount: “))

print(“Processing withdrawal of ₹”, amount)
```

If the input is not an integer, the above code will raise a ValueError. The exception handling mechanism helps to handle such unexpected events without crashing the program.

Exception handling not only strengthens the reliability of your code but also improves user experience by avoiding sudden crashes and offering clear, informative error messages. Properly managed exceptions enable you to log errors, free up resources, or provide alternative options without stopping the entire program. This is crucial in larger applications where a single unaddressed exception might lead to significant problems. By foreseeing potential errors and addressing them with structured exception handling, developers can build more resilient, maintainable, and user-friendly software.

5.1.2 Difference between Syntax Errors and Exceptions

Syntax Error: As the name implies, this error results from incorrect syntax in the code. It results in the program's termination. A syntax error occurs when the code violates the rules of the Python language.

```
# initialize the amount variable

amount = 20000

if(amount > 2999)

print(“You are eligible”)
```

Output:

```
SyntaxError: invalid syntax
```

Exceptions : An exception is an error that occurs during the execution of a program. It is detected during the runtime of the program.

Examples: ZeroDivisionError, ValueError, FileNotFoundError, etc.

When the program is syntactically sound but the code produces an error, exceptions are raised. Although the application continues to run while experiencing this error, the typical course of the program is altered.

```
marks = 10000

a = marks / 0

print(a)
```

Output:

```
ZeroDivisionError: division by zero
```

The attempt to divide a number by zero in the example above triggered the ZeroDivisionError.

5.1.3 Built-in Exceptions in Python

Exception Name	Description
ZeroDivisionError	Raised when a number is divided by zero.
NameError	Raised when a variable is not defined or is used before declaration.
TypeError	Raised when an operation or function is applied to an object of inappropriate type.
ValueError	Raised when a function receives an argument of the correct type but with an inappropriate value.
IndexError	Raised when an index is out of the range of a list, tuple, or string.
KeyError	Raised when a dictionary key is not found.
FileNotFoundError	Raised when trying to open a file that does not exist.
IOError	Raised when an input/output operation fails. (In Python 3, often replaced by OSError)
AttributeError	Raised when an invalid attribute reference is made, or an attribute is not found.
ImportError	Raised when an imported module cannot be found or loaded.
ModuleNotFoundError	Subclass of ImportError, specifically when a module is not found.

IndentationError	Raised when incorrect indentation is used.
SyntaxError	Raised when the Python parser encounters an incorrect syntax.
RuntimeError	Raised when an error is detected that doesn't fall under any other category.
StopIteration	Raised to signal the end of an iterator.
MemoryError	Raised when an operation runs out of memory.
OverflowError	Raised when a numeric operation exceeds the allowed limit.
FloatingPointError	Raised when a floating-point operation fails.
AssertionError	Raised when an assert statement fails.
RecursionError	Raised when the maximum recursion depth is exceeded.

5.1.4 Exception Handling in Python

Exception handling in Python is a powerful feature that allows developers to manage errors without terminating the program. It provides a structured way to detect and respond to exceptional situations that occur during runtime using the try, except, else, and finally blocks. The code that might raise an error is placed inside the try block, and if an exception occurs, it is caught and handled in the except block, allowing the program to continue running or exit cleanly. Python supports both specific and general exception handling, which means you can catch particular exceptions like ZeroDivisionError, ValueError, or use a general Exception class to catch any unexpected error. The optional else block executes only when no exception is raised, and the finally block is used to define clean-up actions that must be executed under all circumstances, such as closing a file or releasing a resource. This approach improves the reliability, readability, and user-friendliness of programs by avoiding crashes.

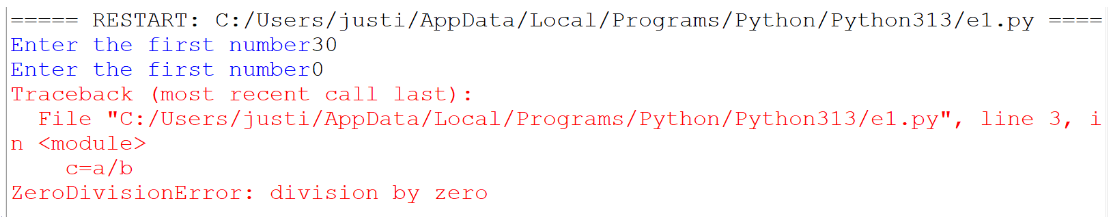
Syntax:

```
try:
   # Block of code
except Exception1:
   # Handler for Exception1
except Exception2:
   # Handler for Exception2
else:
   # Code that runs if no exception occurs
finally:
   # Code that always runs (cleanup actions)
```

Program without Exception Handling

```
a=int(input("Enter the first number"))

b=int(input("Enter the first number"))

c=a/b

print("Result=",c)
```

Output

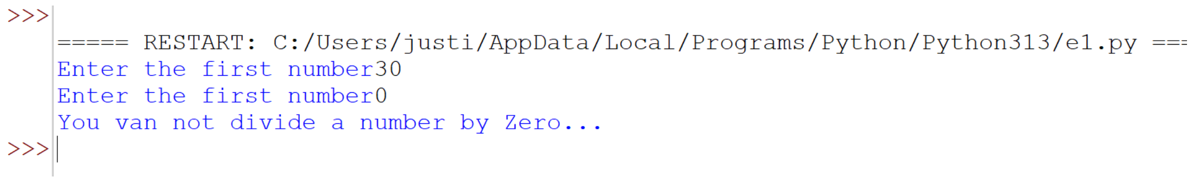


```
===== RESTART: C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py ====
Enter the first number30
Enter the first number0
Traceback (most recent call last):
  File "C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py", line 3, i
n <module>
    c=a/b
ZeroDivisionError: division by zero
```

Program with Exception Handling

```
a=int(input("Enter the first number"))

b=int(input("Enter the first number"))

try:

        c=a/b

        print("Result=",c)

except ZeroDivisionError:

        print("You can not divide a number by zero....")
```

Output



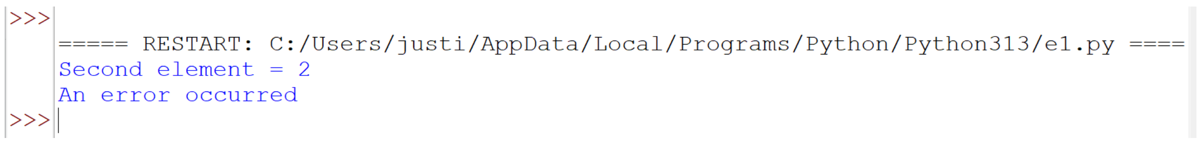
```
>>>
    ===== RESTART: C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py ===
    Enter the first number30
    Enter the first number0
    You van not divide a number by Zero...
>>>
```

5.1.4.1 Try and Except Statement – Catching Exceptions

In Python, exceptions are caught and dealt with using the try and except commands. The try and except clauses are used to contain statements that can raise exceptions and statements that handle such exceptions.

```
a=[1,2,3]

try:

        print("Second element =$d"$(a[i]))

        print("Fourth element =$d"$(a[i]))

except:

print ("An error occurred")
```

Output



```
>>>
    ===== RESTART: C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py ====
    Second element = 2
    An error occurred
>>>
```

The statements that could result in the error are contained inside the try statement in the example above (second print statement in our case). The fourth entry of the list is not accessible in the second print statement, which results in an exception. The except statement then handles this exception.

5.1.4.2 Catching Specific Exception

To provide handlers for various exceptions, a try statement may contain more than one except clause. We may, for instance, add IndexError to the code shown above. The standard syntax for adding certain exceptions –

```
try:
   # statement(s)
except IndexError:
   # statement(s)
except ValueError:
   # statement(s)
def  fun(a):
          if a<4:
               b=a/(a-3)
         if  a>=4:
            #throws NameError
            print("value of b=",b)
try:
    fun(3)
   fun(5)
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

Output

ZeroDivisionError Occurred and Handled

If you comment on the line fun(3), the output will be

NameError Occurred and Handled

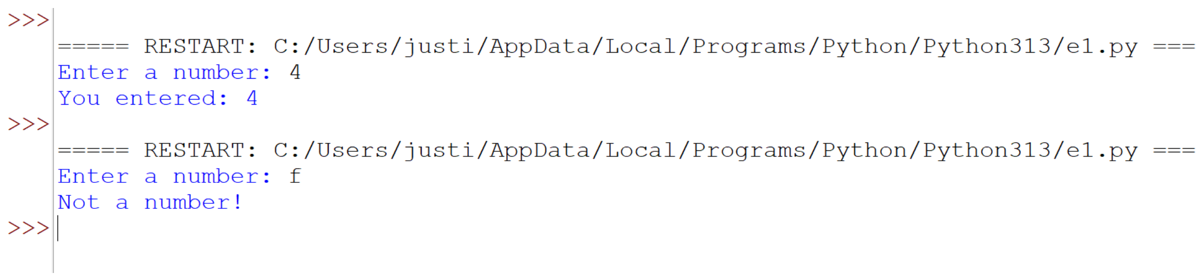
The output above is so because as soon as python tries to access the value of b, NameError occurs.

5.1.4.3 The else Clause

In Python's exception handling mechanism, the else clause is an optional block that can be added after all except blocks in a try-except structure. Its primary purpose is to specify a block of code that should run only if no exceptions are raised in the try block. This enhances code clarity by separating error-handling logic from normal processing logic.

```
try:
    x=int(input("Enter a number:.."))
except ValueError:
    print("Not a number")
else:
    print("You entered")
```

Output



```
>>>
    ===== RESTART: C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py ===
    Enter a number: 4
    You entered: 4
>>>
    ===== RESTART: C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py ===
    Enter a number: f
    Not a number!
>>>|
```

5.1.4.4 Finally Keyword in Python

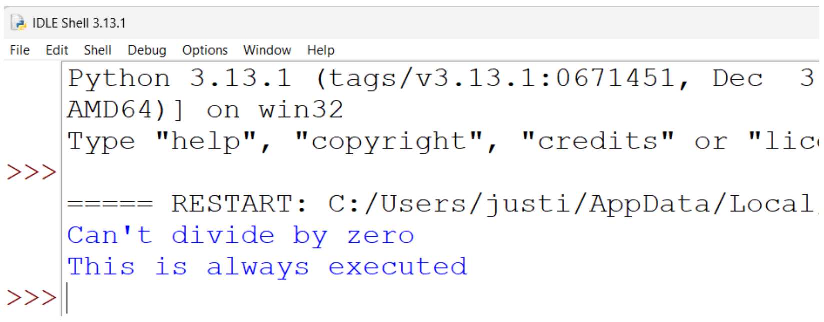
The finally keyword is available in Python, and it is always used after the try and except blocks. The final block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.

Example:

```
try:
    k=5/0
   print(k)
```

```
except ZeroDivisionError:
    print("Can't divide by zero")
finally:
    print("This is always executed")
```

Output:


IDLE Shell 3.13.1

```
Python 3.13.1 (tags/v3.13.1:0671451, Dec  3
AMD64)] on win32
Type "help", "copyright", "credits" or "lic
>>>
===== RESTART: C:/Users/justi/AppData/Local
Can't divide by zero
This is always executed
>>>
```

5.1.5 Raising Exceptions Manually

Python allows you to raise exceptions using the raise keyword. Raising an exception means intentionally causing the program to stop normal execution and instead jump to the nearest exception-handling block (try-except). This is useful when the program encounters invalid data, unexpected input, or any violation of predefined rules.

The user defined-exception can be created using two methods:

1. raise statement

Syntax of raise Statement:

```
raise ExceptionType("Optional error message")
```

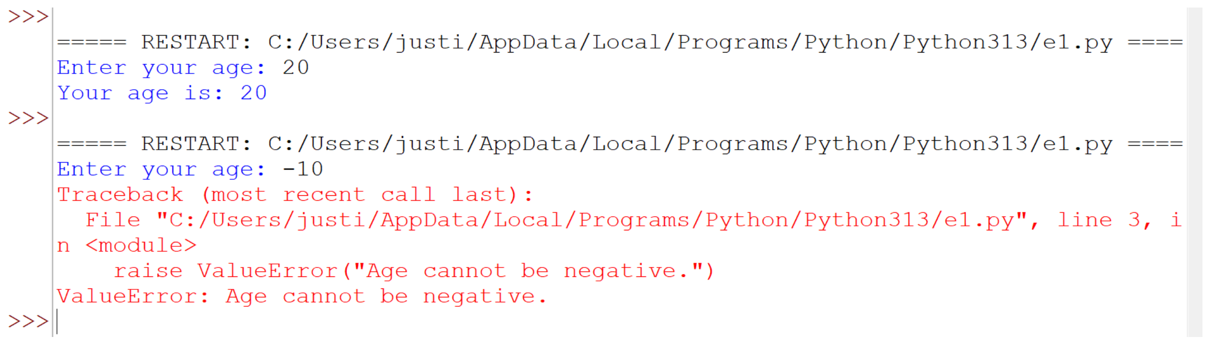
…………………………………………………………………………………………

ExceptionType: Any built-in or user-defined exception class.

Error message (optional): A string message that describes the reason for the exception.

For example, if your program expects a user's age to be a positive number, you can raise a ValueError when a negative value is provided, even if Python itself does not consider it a built-in error.

```
age=int(input("Enter your age: "))
if age<0:
    raise ValueError("Age cannot be negative.")
print("Your age is ",age)
```



```
>>>
===== RESTART: C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py ====
Enter your age: 20
Your age is: 20
>>>
===== RESTART: C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py ====
Enter your age: -10
Traceback (most recent call last):
  File "C:/Users/justi/AppData/Local/Programs/Python/Python313/e1.py", line 3, i
n <module>
    raise ValueError("Age cannot be negative.")
ValueError: Age cannot be negative.
>>>
```

2. assert statement

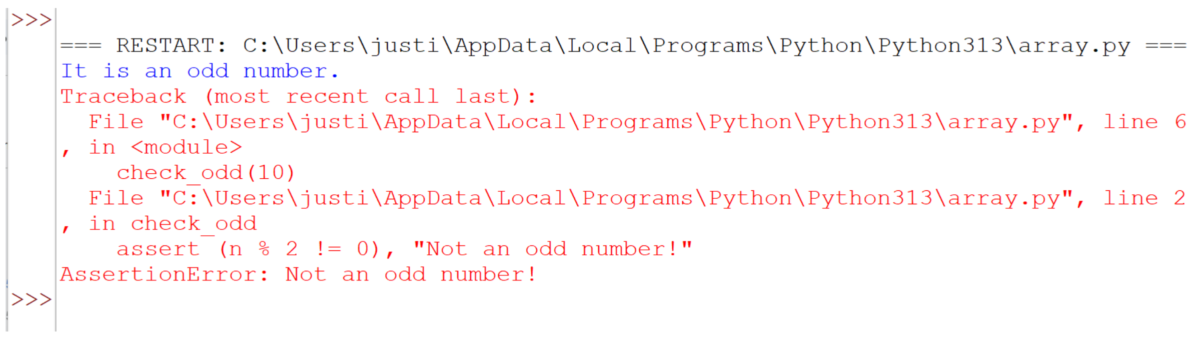
The assert statement in Python is used to verify whether a specific condition holds true. If the condition evaluates to false, an exception is triggered. Typically, this statement is placed at the start of a function or right after calling a function to ensure the input or result is valid. The basic syntax of the assert statement is:

assert condition[, message]

When Python encounters an assert statement, it checks the condition provided right after the assert keyword. If the condition is not satisfied (i.e., it evaluates to false), Python raises an AssertionError, which can be caught and managed just like any other exception.

```
def check_odd(n):
        assert(n%2!=0), "Not an odd number!"
         print("It is an odd number.")
check_odd(7)
check_odd(10)
```

Output



```
>>>
=== RESTART: C:\Users\justi\AppData\Local\Programs\Python\Python313\array.py ===
It is an odd number.
Traceback (most recent call last):
  File "C:\Users\justi\AppData\Local\Programs\Python\Python313\array.py", line 6
, in <module>
    check_odd(10)
  File "C:\Users\justi\AppData\Local\Programs\Python\Python313\array.py", line 2
, in check_odd
    assert (n % 2 != 0), "Not an odd number!"
AssertionError: Not an odd number!
>>>
```

5.1.5 Advantages of exception handling in python

- Prevents program crashes by handling errors without stopping execution.
- Improves code clarity by separating error-handling logic from regular code.
- Manages unexpected runtime errors such as division by zero or file not found.

- Allows custom error messages to provide user-friendly feedback.
- Supports the creation and use of custom exceptions for specific error types.
- Ensures the rest of the program can continue running after an error is handled.
- Helps in debugging by providing detailed error tracebacks.
- Increases program robustness and reliability by handling faults effectively.

Recap

- **Python errors are categorized into five types**: Compile Time Errors, Run Time Errors (exceptions), Logical Errors, Syntax Errors, and Semantic Errors. Each has distinct causes and effects on program execution.
- **Syntax Errors** occur due to incorrect Python syntax, such as missing colons, wrong indentation, or invalid expressions. They are detected before program execution and stop the code from running.
- **Exceptions** are errors that occur during the program's execution, even if the syntax is correct. These include situations like dividing by zero, accessing invalid indexes, or providing incorrect input types.
- **Logical Errors** arise from mistakes in the logic or flow of the program. These errors don't stop the program but result in incorrect output.
- **Semantic Errors** are meaning-related mistakes—code that is syntactically correct but does not do what the programmer intended.
- In Python, when an error occurs during execution, an **exception is raised**, which interrupts the normal flow unless it is properly handled by the programmer.
- **Exception objects** are created when an error occurs, and Python allows developers to catch and respond to these exceptions using structured blocks (try, except, etc.).
- The try block is used to wrap code that might cause an exception. If an error occurs, Python jumps to the matching except block.
- Multiple except blocks can be used to handle **specific exceptions**, such as IndexError, ValueError, or ZeroDivisionError, enabling more controlled and meaningful responses to different types of errors.
- The else clause, when used with try-except, runs **only if no exception** occurs in the try block. It helps separate successful execution logic from error-handling code.
- The finally block runs **regardless of whether an exception occurred** or not. It's typically used for releasing resources or performing mandatory clean-up actions.

- Python allows **raising exceptions manually** using the raise keyword. This helps enforce specific conditions, such as raising a ValueError when the input is not acceptable.
- The assert statement checks whether a condition is True; if not, it raises an AssertionError. It's often used to validate assumptions during development.
- **Built-in exceptions** in Python include ZeroDivisionError, NameError, TypeError, ValueError, FileNotFoundError, IndexError, KeyError, and many more, each with a specific purpose and usage.
- Exception handling in Python improves **program reliability**, prevents crashes, makes error messages more user-friendly, and helps in building clean, maintainable, and professional-level software.

Objective Type Questions

1. What type of error occurs when the code violates Python's grammatical rules?
2. What is raised during the execution of a program when an unexpected error occurs?
3. Which keyword is used in Python to handle exceptions?
4. What error occurs when a number is divided by zero?
5. Which block is always executed, whether or not an exception occurs?
6. Which keyword is used to raise an exception manually?
7. Which error is raised when an `assert` condition fails?
8. What is the correct block used to handle code that might cause an error?
9. Which error occurs when a specified dictionary key is not found?
10. What error occurs when an undefined variable is accessed?
11. What is the error called when an operation or function is applied to an object of inappropriate type?
12. Which error is raised when a list index is out of range?
13. Which statement is used to check a condition and raise an error if it fails during development?

14. What error is raised when a required module cannot be found?

Answers to Objective Type Questions

1. SyntaxError
2. Exception
3. Except
4. ZeroDivisionError
5. Finally
6. Raise
7. AssertionError
8. Try
9. KeyError
10. NameError
11. TypeError
12. IndexError
13. Assert
14. ModuleNotFoundError

Assignments

1. Explain the different types of errors in Python programming with suitable examples.
2. What are exceptions in Python? How does Python handle exceptions using try, except, else, and finally blocks? Explain with examples.
3. Write a detailed note on built-in exceptions in Python. Mention at least 8 common exceptions with a brief explanation of each.
4. What is the difference between Syntax Errors and Exceptions in Python? Illustrate your answer with appropriate code examples.
5. Explain how to raise exceptions manually in Python. What is the role of the raise and assert statements in exception handling? Provide examples.

Reference

1. Clark, W. E. (2025). Python Exception Handling Made Easy: A Practical Guide with Examples (1st ed.).

2. Lutz, M., & Ascher, D. (2023). Exception Basics. In Learning Python (6th ed., Chapter 33). O'Reilly Media.

3. Pearson, J. (2020). Introduction to Python Programming and Data Structures (3rd ed., Chapter 13: Files and Exception Handling). Pearson.

4. Bhasin, H. (2023). Python Programming Using Problem Solving (Chapter 15: Exception Handling). Mercury Learning and Information.

Suggested Reading

1. Ceder, N. (2021). The Quick Python Book (3rd ed.). Manning Publications.

2. Zelle, J. M. (2017). Python Programming: An Introduction to Computer Science (3rd ed.). Franklin, Beedle & Associates.

3. Downey, A. (2015). Think Python: How to Think Like a Computer Scientist (2nd ed.). O'Reilly Media.

4. Python Official Documentation. (n.d.). Errors and Exceptions – Python 3.x Docs. https://docs.python.org/3/tutorial/errors.html

Unit 2
Debugging techniques and tools

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- understand the importance of debugging in ensuring the correctness and reliability of Python programs.
- use the pdb module to pause program execution, inspect variables, and step through code interactively.
- identify and fix common errors such as syntax errors, logical errors, and runtime exceptions.
- apply exception handling techniques using try, except, and related blocks to manage errors gracefully.
- Compare different debugging methods such as print statements, logging, pdb, and IDE-based tools.

Prerequisites

Imagine a situation where a student writes a Python program to calculate grades. The code appears to be correct, but when the program runs, it displays unexpected results, such as showing zero percent for a student who scored well. In such cases, scanning the code repeatedly may not help identify the problem easily. Instead of using multiple print statements or making random guesses, a debugger like Python's pdb can be used to pause the execution, examine the values of variables, and understand the exact flow of the program. This helps to identify the exact point of error and correct it effectively. Debugging is a crucial skill that supports real-time application development and ensures programs behave as expected.

This chapter explains the concept and importance of debugging along with various techniques used in Python such as print statements, exception handling, logging, and the use of the pdb module. The topic covers how to use different pdb commands like p, n, c, and q to step through code, inspect variable values, and control the execution flow. Learning these techniques helps to make the code more accurate, readable, and manageable. Debugging improves the quality of software, prevents unexpected crashes, and saves valuable time. After learning this topic, the skill to detect and fix errors becomes stronger and helps in writing efficient and dependable Python programs. Debugging also sharpens logical thinking and problem-solving ability which are essential for programming success.

Key words

Debugging, Python, pdb, Breakpoints, Exception Handling, Logging

Discussion

5.2.1 Introduction to Debugging in Python

Debugging is the process of locating, understanding, and fixing errors or bugs in a software program. In Python, debugging plays a crucial role in ensuring a program works as intended. Rather than waiting until a user reports an issue, debugging allows developers to detect errors early, trace their origins, and correct them while maintaining the integrity and logic of the overall program.

5.2.1.1 Importance of debugging

- ♦ Ensures that each part of the program runs correctly by validating the logic and flow of execution.
- ♦ Enhances the reliability and readability of the code by systematically identifying and fixing both minor and critical bugs.
- ♦ Helps detect and resolve issues that may cause the program to behave unexpectedly or terminate abruptly.
- ♦ Makes the codebase easier to manage, allowing future modifications, improvements, and debugging to be performed more efficiently.
- ♦ Leads to a more stable and user-friendly application by preventing common errors and improving the overall software experience.

5.2.2 Debugging Techniques

Effective debugging is essential for identifying and fixing issues in your Python code. Here are some commonly used debugging techniques:

5.2.2.1 Print Statements

The simplest form of debugging is by inserting print() statements in your code to trace variable values or program flow.

```
def  add (a,b):
     print(f"a:   {a}, b:{b}")
      return a+b
result=add(5,10)
print(f"Result: {result}")
//This  print() helps to verify that the values passed and returned are correct.
```

5.2.2.2 Using the pdb Module

pdb is Python's built-in debugger. It allows pause execution, inspect variables, and step through code. This is **very useful**, especially in **large programs** where bugs can be hidden in complex logic.

```
Import pdb

def divide(a,b):

      pdb.set_trace() #pause execution here

     return a/b

result=divide(10,2)

print(result)
```

When you run this program, Python will execute the function divide(10, 2). But inside the divide function, we wrote pdb.set_trace(). This stops the program there and enters the debugging mode. Now see something like this in your terminal:

```
> File "myfile.py", line 4, in divide
  pdb.set_trace()
(Pdb)
```

This means the debugger has paused the program, and interacts with the program live. Here are some common commands type after (Pdb):

- p a → This will print the value of variable a. Output: 10
- p b → This will print the value of variable b. Output: 2
- n → This means the next line. It tells Python to go to the next line of code (like return a / b)
- c → Continue running the program until it finishes or hits another breakpoint.
- q → Quit the debugger. It will stop the program completely.

5.2.2.3 Exception Handling

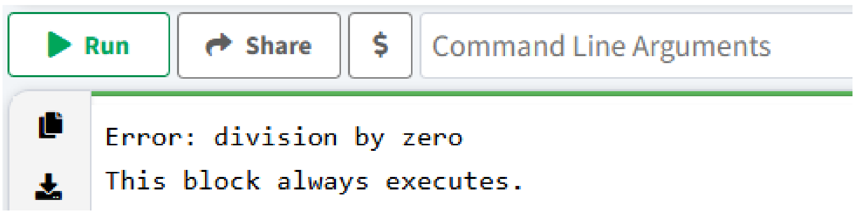
Instead of allowing the program to crash, exceptions allow graceful handling of errors.

```
try:

   result=10/0

except ZeroDivisionError as e:

   print("Error: {e}")

else:

   print("An error occurred")
```

finally:

print("This block always executes.")

Here, except catches the division error and allows the program to continue.



```
Error: division by zero
This block always executes.
```

5.2.2.4 Logging

Logging records events during code execution. It is better than print statements for production.

Import logging

Logging.basicConfig(level.logging.DEBUG)

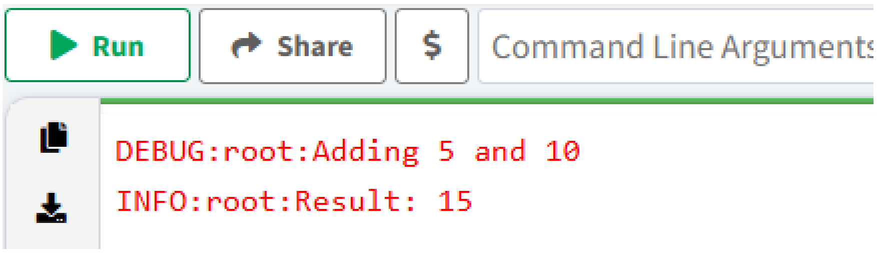
def add(a,b):

    logging.debug("Adding{a} and {b}")

     return a+b

result=add(5,10)

logging.info(f"Result: {result}")



```
DEBUG:root:Adding 5 and 10
INFO:root:Result: 15
```

5.2.2.5 Using IDE Debuggers

IDEs like PyCharm, VS Code, and Spyder come with graphical debuggers. Features include:

- Breakpoints
- Step-by-step execution
- Variable inspection
- Call stack visualization

5.2.3 pdb (Python Debugger)

pdb (Python Debugger) is Python's built-in interactive debugging tool. It provides developers with the ability to inspect the state of a program while it is running, allowing for efficient identification and correction of errors. The pdb (Python Debugger) is a native debugging tool in Python, crafted to assist developers in finding and resolving errors in their code. It offers an interactive interface to analyze the program's state while it is running, enabling features such as step-by-step execution, variable inspection, and runtime code adjustments.

5.2.3.1 Key Features of pdb

- Setting Breakpoints: Enables you to halt the execution of your program at designated locations to review the current state of the application.
- Stepping through code: Allows you to run code one line at a time, helping you track the flow of execution and pinpoint where issues arise.
- Inspecting Variables: Provides the ability to view the values of variables at various stages of execution to track their changes.
- Call stack view: Grants access to the call stack, letting you observe the chain of function calls leading up to the current line of code.
- Interactive Controls: Includes a set of commands to control the flow of execution, set breakpoints, and navigate through the code interactively.

5.2.3.2 Key Concepts with examples

1. Breakpoints:

- Objective: Interrupt the program at specified points to examine its state.
- Application: You can set breakpoints using commands like break <line number> or break <function name>.

```
import pdb

def add(a, b):

    return a + b

pdb.set_trace()  # Sets a breakpoint

result = add(5, 10)

print(result)
```

2. Step Execution:

- Objective: Execute your code one line at a time to monitor its flow and detect issues.
- Application: Use the step (or s) command to move through each line of code, entering functions as they are called.

```
    def divide(a, b):
        import pdb; pdb.set_trace()
        return a / b
result = divide(10, 2)
print(result)
```

3. Variable Inspection:

- Objective: Check the values of variables at different points in the program to understand their changes.
- Application: Use the print (or p) command to display the value of variables

```
def multiply(a, b):
    import pdb; pdb.set_trace()
    result = a * b
    return result
print(multiply(3, 4))
```

4. **Interactive Commands:**

- Objective: Control the program's execution, set and manage breakpoints, and navigate through the code interactively.
- Application: Commands like continue (or c) to resume execution, next (or n) to step over lines, and quit (or q) to exit the debugger are used.

```
def main():
  import pdb; pdb.set_trace()
    x = 10
    y = 20
    result = x + y
    print(result)
main()
```

5.2.3.3 Basic pdb Commands

- break or b: The purpose of the break command in pdb is to set a breakpoint at a specific line number or function. For example, break 12 sets a breakpoint at line 12, and break my_function sets it at the start of the function named my_function.
- continue or c: The purpose of the continue command in pdb is to resume the program's execution after it has been paused. It runs the code until the next breakpoint is encountered or the program finishes.

- step or s: The purpose of the step command in pdb is to move into the next line of code, including entering into function calls. This helps to closely observe the internal execution of functions and understand the flow in detail.
- next or n: The purpose of the next command in pdb is to execute the current line and move to the next line in the same function without stepping into any called functions. It is useful for quickly moving through code while staying within the current function scope.
- list or l: The purpose of the list command in pdb is to display the surrounding lines of source code to provide context for the current execution point. Using list or list 10,20 helps in viewing specific line ranges and understanding the code structure around the breakpoint.
- print or p: The purpose of the print command in pdb is to show the current value of a variable or the result of an expression during debugging. For example, print my_variable or print my_variable + 5 helps to check the data and trace logic errors.
- quit or q: The purpose of the quit command in pdb is to exit the debugger and stop the execution of the program immediately. It is used when debugging is complete or if the user wants to terminate the session early.
- where or w: The purpose of the where command in pdb is to display the current call stack, showing the sequence of function calls that led to the current point in the program. This helps in understanding the execution path and identifying where the error originated.
- return: The purpose of the return command in pdb is to resume execution until the current function finishes and returns to its caller. This is useful when the internal steps of a function are not needed for inspection, and focus is on the result.
- disable: The purpose of the disable command in pdb is to temporarily deactivate a specified breakpoint without removing it from the list. For example, disable 1 turns off breakpoint number 1 but allows it to be re-enabled later.
- enable: The purpose of the enable command in pdb is to reactivate a breakpoint that was previously disabled. For example, enable 1 re-enables breakpoint number 1 so that it becomes active again during execution.

5.2.3.4 How pdb Debugging Works in Python

pdb (Python Debugger) is a tool that provides an interactive environment for examining and controlling the execution of Python code. Here’s a detailed explanation of how pdb operates:

1. Inserting a Breakpoint:

To start debugging with pdb, you need to insert a breakpoint in your code. This is done using the statement import pdb; pdb.set_trace(). When the execution reaches this point, the debugger halts, and you enter the pdb interactive mode.

Example:

```
def divide(a, b):
import pdb; pdb.set_trace()  # Execution will pause here
return a / b
```

2. Starting the Debugging Session

Run the Python script as you normally would. The execution will pause at the line where pdb.set_trace() is located, and you will see the pdb prompt, allowing you to interact with the debugger.

Running the Program:

```
python myscript.py
```

3. Using pdb Commands

Once the debugger is active, you can use various commands to control the execution flow and inspect the program's state:

print (or p): Displays the current value of a variable or the result of an expression.

Example:

```
(Pdb) p a
10
```

step (or s): Executes the current line of code and steps into any called functions, allowing you to examine each step closely.

Example:

```
(Pdb) s
```

4. next (or n): Moves to the next line within the current function, bypassing any function calls on the current line.

Example:

```
(Pdb) n
```

5. continue (or c): Resumes the execution of the program until it hits the next breakpoint or finishes.

Example:

```
(Pdb) c
```

6. list (or l): Shows the source code around the current line to provide context.

Example:

```
(Pdb) l
```

7. where (or w): Displays the call stack, showing the sequence of function calls that led to the current point in the code.

Example:

```
(Pdb) w
```

8. quit (or q): Exits the debugger and terminates the program.

5.2.3.5 Practical Example with pdb

```
Import pdb

def fxn(n):

    for i  in range(n):

        print("Hello!", i+1)

pdb.ser_tace()

fxn(5)
```

This program prints "Hello!" followed by a number from 1 to n.

Before calling the function fxn(5), it pauses the program using the Python Debugger (pdb).

pdb.set_trace() lets you pause execution, inspect variables, and step through the code before fxn(5) is executed.

This program prints "Hello!" followed by a number from 1 to n. Before calling the function fxn(5), it pauses the program using the Python Debugger (pdb). pdb.set_trace() lets you pause execution, inspect variables, and step through the code before fxn(5) is executed. Function fxn(n): A simple function that loops n times (in this case, 5). On each loop, it prints: Hello! 1, Hello! 2, ..., Hello! 5. pdb.set_trace() This is a breakpoint. The program pauses here before calling the function. Then see the (Pdb) prompt in the terminal. While at the (Pdb) prompt: n – Go to the next line. c – Continue until the next breakpoint or program end. p n – Print value of the variable n (if available in scope). q – Quit the debugger. After pressing c, the program resumes and: Enters the function fxn(5)

```
Hello! 1

Hello! 2

Hello! 3

Hello! 4

Hello! 5
```

To practice basic debugging using the pdb module. To learn how to pause the program, inspect variables, and control execution before running the main logic. It helps in checking what happens before the function is executed. It verify that fxn(5) is called correctly.

5.2.3.6 Advantages of using pdb debugger

- It allows you to run the program line by line.
- To check the values of variables during execution.
- It helps you find the exact location of errors.
- To pause the program at any point using pdb.set_trace().
- It reduces the need for multiple print statements.
- It is built into Python, so no extra installation is needed.
- It helps in understanding how the code works step by step.
- It is useful for debugging complex logic or large programs.

Recap

- Debugging in Python is essential to ensure a program runs as intended without unexpected behavior.
- It helps in identifying logical, runtime, and syntax errors early in the development process.
- Effective debugging improves the overall quality and performance of the software.
- Using print() statements is a quick method to observe variable values and program flow.
- However, relying only on print() becomes difficult in large programs with nested logic.
- The pdb module offers an interactive way to pause execution and examine code behavior in detail.
- Developers can set breakpoints using pdb.set_trace() to stop the program at a specific line.
- At the (Pdb) prompt, commands like p (print), n (next), c (continue), and q (quit) are used to navigate and inspect.
- Variables can be inspected and expressions evaluated during a paused state, making it easier to identify issues.

- ♦ You can step into functions to observe how data is passed and processed internally.
- ♦ Exception handling using try-except blocks allows for graceful handling of errors like division by zero.
- ♦ This prevents the program from crashing and gives the user a proper message or alternative outcome.
- ♦ Logging is more powerful than print statements as it keeps records of events, errors, and flow for later analysis.
- ♦ Log files help debug issues even after the program has been deployed.
- ♦ Modern IDEs offer built-in graphical debuggers, making debugging more visual and intuitive.
- ♦ Features like line-by-line execution, variable watch windows, and call stack views simplify complex debugging.
- ♦ The pdb debugger is lightweight, requires no setup, and is especially useful when a GUI debugger is not available.
- ♦ Commands like break, step, next, continue, and where provide full control over the debugging session.
- ♦ A practical example with a loop demonstrated how pdb can pause before function execution to examine logic.
- ♦ This helps verify input values, function calls, and outputs before the main logic runs.
- ♦ Mastering debugging techniques improves developer efficiency and helps maintain large codebases.
- ♦ Overall, pdb is a powerful, flexible, and accessible tool for both beginners and experienced Python developers.

Objective Type Questions

1. What is the process of identifying and fixing errors in a program called?
2. Which module in Python is used for interactive command-line debugging?
3. When does the program pause during pdb debugging?
4. How can you print the value of a variable in pdb?
5. Why is logging preferred over print statements in production code?

6. Where do you insert pdb.set_trace() in the program?
7. Who uses the debugger to step through code and inspect variables?
8. What command resumes program execution until the next breakpoint?
9. How do you exit the pdb debugger?
10. Which command in pdb steps into a function call?
11. What is shown using the list command in pdb?
12. Why is exception handling important in a program?
13. How is a breakpoint created in pdb?
14. When should you use try-except blocks in Python?
15. What prompt appears when the pdb debugger is active?

Answers to Objective Type Questions

1. Debugging
2. pdb
3. Breakpoint
4. print
5. Reliability
6. Code
7. Developer
8. continue
9. quit
10. step
11. Code
12. Safety
13. set_trace
14. Error
15. (Pdb)

Assignments

1. Explain the importance of debugging in Python. How does it help in improving program quality?
2. What are the different debugging techniques in Python? Describe each technique with examples.
3. Describe the role of the pdb module in Python debugging.
4. Write a detailed note on how to use the pdb module with appropriate examples. Include commands and their usage.
5. Explain the step-by-step process of debugging a Python program using pdb.set_trace(). What happens after execution pauses?
6. Explain how to use basic pdb commands like break, step, next, continue, list, and quit.

Reference

1. Clark, W. E. (2025). Python Exception Handling Made Easy: A Practical Guide with Examples (1st ed.).
2. Lutz, M., & Ascher, D. (2023). Exception Basics. In Learning Python (6th ed., Chapter 33). O'Reilly Media.
3. Pearson, J. (2020). Introduction to Python Programming and Data Structures (3rd ed., Chapter 13: Files and Exception Handling). Pearson.
4. Matthes, E. (2023). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming* (3rd ed.). No Starch Press.
5. Lutz, M. (2021). *Learning Python* (5th ed.). O'Reilly Media.
6. Slatkin, B. (2020). *Effective Python: 90 Specific Ways to Write Better Python* (2nd ed.). Addison-Wesley Professional.

Suggested Reading

1. Ceder, N. (2021). The Quick Python Book (3rd ed.). Manning Publications.
2. Slatkin, B. (2020). *Effective Python* (2nd ed.). Addison-Wesley Professional.
3. Zelle, J. M. (2017). Python Programming: An Introduction to Computer Science (3rd ed.). Franklin, Beedle & Associates.
4. https://docs.python.org/3/library/logging.html
5. https://docs.python.org/3/library/pdb.html
6. https://realpython.com/python-debugging-pdb/

Unit 3
Database Programming in Python

Learning Outcomes

After the successful completion of the unit, the learner will be able to:

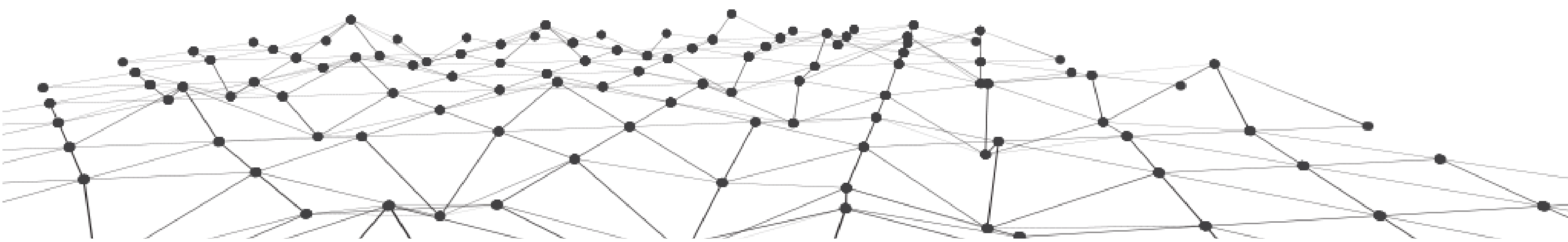
- define what a database is and list common types of databases used with Python.
- identify the basic components of an SQL query such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- explain how Python can be connected to a database using database connectors like `MySQL connector`.
- describe the steps involved in executing an SQL query from a Python program.
- recognize the purpose of each basic SQL operation and its role in managing data through Python scripts.

Prerequisites

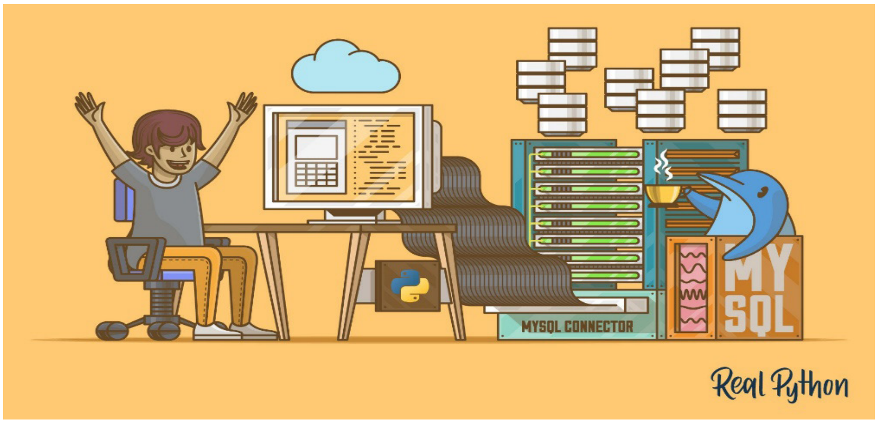
Before starting with database programming in Python, learners should have a basic understanding of how data is stored and organized in real-life scenarios. For example, consider a library system where books, members, and borrowing records must be tracked. This kind of structured data is best handled using a ***database***, which stores information in tables made up of rows and columns. Knowing what a database is, and why it's useful, helps learners understand the importance of storing, retrieving, and managing data efficiently.

A basic knowledge of *SQL (Structured Query Language)* is also important. SQL is the language used to interact with most databases. It allows you to perform operations like retrieving data with `SELECT`, adding new data with `INSERT`, modifying records with `UPDATE`, and removing unwanted data using `DELETE`. For instance, if you want to find all books borrowed by a particular student, you would use an SQL `SELECT` query to get the information from the database table.

Lastly, learners should be comfortable with *Python programming fundamentals* such as variables, functions, conditionals, and loops. Since this unit focuses on connecting Python with a database, it is important to understand how to write basic Python pro-



grams and how Python can interact with external files or systems. For example, using Python to automatically update the stock of products in a supermarket database whenever a sale is made demonstrates how Python and SQL work together in real-world applications.



Key words

MySQL, Import, Cursor, Connectors, SQLite

Discussion

5.3.1 Introduction to Databases in Python

Databases are essential tools in the world of software development, used to store, manage, and retrieve data efficiently. In today's data-driven world, applications often rely on structured databases to organize information, such as customer details, product inventories, or user activities. Python, being one of the most widely-used programming languages, offers built-in libraries and modules that allow developers to interact seamlessly with various types of databases like SQLite, MySQL, and PostgreSQL. Understanding how to connect Python programs with databases is a critical step for building dynamic and data-centric applications.

Python provides robust support for database programming through modules like `sqlite3`, which allows developers to execute SQL commands directly from Python scripts. With these tools, one can perform fundamental database operations such as creating tables, inserting records, updating values, and deleting unwanted data. For example, using Python to manage a student database system makes it easy to automate tasks like registering new students or updating marks, all while keeping the data organized and searchable. This integration of Python and SQL enables powerful and flexible data manipulation capabilities for both beginners and professionals.

In real-world applications, database programming in Python is widely used in web development, data analysis, and automation tasks. Consider a small online bookstore where customer orders and inventory need to be tracked. By using Python to connect to a database, the system can automatically update stock levels, retrieve customer order history, and generate sales reports. Learning to work with databases in Python not only builds a foundation for handling real data but also prepares students for more advanced topics such as data modeling, security, and web application backends.

5.3.2 SQL basics

SQL (Structured Query Language) is a special-purpose language used to interact with databases. It allows you to store, retrieve, update, and delete data from a relational database. Python, a popular and easy-to-learn programming language, provides built-in support to work with databases using SQL. Learning how to use SQL with Python helps beginners build real-world applications that can manage and analyze data effectively.

SQL is used to perform operations on the data stored in relational databases. A relational database stores data in tables which are like spreadsheets made up of rows and columns. Some common SQL commands are shown in the table 5.3.1 below:

Table 5.3.1 Common SQL commands

Sql Command	Purpose	Example
SELECT	Read data from a table	SELECT * FROM students;
INSERT	Add new data to a table	INSERT INTO students VALUES ('John', 20);
UPDATE	Modify existing data	UPDATE students SET age = 21 WHERE name = 'John';
DELETE	Remove data from a table	DELETE FROM students WHERE age < 18;

Python programs often need to work with stored data. For example, an app that stores users' names, login info, or test scores needs a database.

Understanding SQL basics in Python is an essential step for any beginner who wants to work with data. SQL helps manage the data, and Python gives the tools to automate and analyze it. Once you understand basic SQL commands and how to use them in Python, you'll be ready to build applications like student databases, address books, inventory systems, and more.

5.3.3 Connecting Python with databases

Databases are used to store and manage large amounts of data efficiently. Python, being a powerful and flexible programming language, can connect to different types of databases to access and manipulate data. This connection is made using database connectors or APIs, which act as bridges between Python and the database. Once connected, Python can send SQL commands to create tables, insert data, retrieve data, and much more.

Types of Databases and Connectors

To work with data effectively in Python, it is important to understand the different types of databases available and how to connect to them. A database is a structured collection of data, and Python can interact with various kinds of databases using special tools called connectors or libraries.

Each database system has its own features, performance level, and use cases. Some databases are lightweight and store data in a single file (like SQLite), while others are powerful systems that support large-scale, multi-user environments (like MySQL, PostgreSQL, and Oracle). Python provides different libraries for connecting to each of these databases, allowing developers to run SQL queries, manage data, and build data-driven applications.

In this section, you will learn about the most commonly used relational databases and the Python connectors required to communicate with them. Understanding these options will help you choose the right database and connector for your project. Python supports connections to various relational databases using different libraries (Table 5.3.2):

Table 5.3.2 Types of Databases and Connectors

Database Type	Python Library	Description
SQLite	sqlite3 (built-in)	Lightweight, file-based database
MySQL	mysql.connector, PyMySQL	Popular open-source database
PostgreSQL	psycopg2	Advanced open-source relational DB
Oracle	cx_Oracle	Connects Python to Oracle DB
SQL Server	pyodbc	Connects to Microsoft SQL Server

5.3.3.1 Using SQLite with Python

SQLite is a lightweight database that stores data in a local file. It requires no separate server setup, making it perfect for beginners and small projects. The step-by-step procedure are:

1. Import the database library: To use a database in Python, you first need to import the appropriate library. For SQLite, Python provides a built-in module called `sqlite3`, so there is no need to install anything extra.

```
 import sqlite3
```

2. Connect to the Database: This line creates a connection to a database file named `school.db`. If the file doesn't exist, it will be created automatically. The `conn` object now acts as a link between your Python program and the database.

```
conn = sqlite3.connect('school.db')  # Creates 'school.
db' if it doesn't exist
```

3. **Create a Cursor:** A cursor is used to send SQL commands to the database and fetch

results. You need a cursor object to execute queries like `SELECT`, `INSERT`, `UPDATE`, etc.

```
cursor = conn.cursor()
```

4. Execute SQL Commands: This SQL command creates a table named `students` if it doesn't already exist. The table will store each student's ID, name, and age. The `execute()` function is used to send this SQL statement to the database.

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    age INTEGER
)
' ' ')
```

5. Insert Data into the Table: This command adds a new record to the `students` table. The question marks `?` are placeholders used in parameterized queries to safely insert values and prevent SQL injection attacks.

```
cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)", ('Alice', 
20))
```

6. Fetch Data: This command retrieves all records from the `students` table. The `fetchall()` function returns all the rows, which can be printed or processed further in the program.

```
cursor.execute("SELECT * FROM students")
print(cursor.fetchall())
```

7. Commit changes: After making changes like inserting or updating records, you need to **commit** those changes to the database. This makes sure your changes are saved.

```
conn.commit()
```

8. Close the Connection: Closing the connection releases the resources and ensures that no more commands are sent to the database. It's a good practice to always close the connection when you're done.

```
conn.close()
```

Connecting Python to databases is a vital step in building real-world applications that rely on data storage and retrieval. SQLite is ideal for learning and simple projects, while other databases like MySQL or PostgreSQL are used in larger systems. With basic knowledge of database connections, you can create robust Python programs that interact with data efficiently.

5.3.3.2 Using MySQL with Python

MySQL is a popular open-source relational database management system used in many web and enterprise applications. It is known for its speed, reliability, and ability to handle large volumes of data. Python can easily connect to a MySQL database using special libraries that allow the execution of SQL commands directly from Python programs. This helps in building real-world applications where data storage and retrieval are essential. Before connecting to MySQL from Python, make sure the following are installed:

1. **MySQL Server**: You need MySQL installed on your system or use a remote MySQL server.

2. **MySQL Connector for Python**: This is the official library provided by Oracle to connect Python with MySQL.

Steps to Use MySQL with Python

Connecting Python to a MySQL database (Table 5.3.3) involves a series of straightforward steps. These steps help you establish communication between your Python program and the MySQL server so that you can send SQL commands, retrieve data, and manage your database efficiently.

Starting with importing the right library, you will learn how to create a connection, execute SQL queries using a cursor, insert and retrieve data, and properly close the connection when finished. Each step builds upon the previous one to form a complete workflow that makes database programming with Python simple and effective.

In the following section, you will explore these essential steps in detail with examples, helping you gain confidence to work with MySQL databases in your Python applications. Below are the basic steps to connect Python with a MySQL database:

1. Import the MySQL Connector: This imports the `mysql.connector` module which contains all the functions needed to connect and interact with a MySQL database.

```
import mysql.connector
```

2. Establish a Connection: To work with a MySQL database in Python, the first important step is to establish a connection between your Python program and the MySQL server. This connection acts as a bridge that allows your program to send SQL commands and receive data from the database. Python uses the `mysql.connector` library to create this connection. When establishing the connection, you need to provide some key information:

- `host`: The location of the MySQL server (use "`localhost`" for your computer).
- `user`: Your MySQL username (commonly "`root`").
- `password`: The password for your MySQL account.
- `database`: The name of the database you want to use.

```
  conn = mysql.connector.connect(
    host=”localhost”,
    user=”root”,
    password=”your_password”,
    database=”school”

)
```

Make sure the database `school` already exists. If the connection is successful, the variable `conn` will hold the connection object, which you can use to interact with the database. If there is any problem (like wrong password or database name), an error will be raised.

3. Create a Cursor: The cursor is used to send SQL commands and fetch data from the database.

```
  cursor = conn.cursor()
```

4. Create a Table: This SQL command creates a students table with id, name, and age columns.

```
cursor.execute(‘’’

CREATE TABLE IF NOT EXISTS students (

    id INT AUTO_INCREMENT PRIMARY KEY,

    name VARCHAR(50),

    age INT

)
‘’’)
```

5. Insert Data: The `%s` placeholders are used for parameterized queries, which help prevent SQL injection.

```
sql = “INSERT INTO students (name, age) VALUES (%s, %s)”

val = (“Alice”, 20)

cursor.execute(sql, val)
```

6. **Retrieve Data:** This code retrieves all records from the `students` table and prints each one.

```
cursor.execute(“SELECT * FROM students”)

result = cursor.fetchall()

for row in result:

print(row)
```

7. Commit and Close: `commit()` saves any changes made to the database and `close()` safely ends the connection.

```
conn.commit()
conn.close()
```

Table 5.3.3 Summary Table of Steps to Use MySQL with Python

Step	Action	Function
1	Import MySQL library	`import mysql.connector`
2	Connect to MySQL	`mysql.connector.connect()`
3	Create a cursor	`conn.cursor()`
4	Execute SQL (create, insert)	`cursor.execute()`
5	Fetch data	`cursor.fetchall()`
6	Commit changes	`conn.commit()`
7	Close connection	`conn.close()`

Using MySQL with Python allows you to build robust applications with powerful data storage and retrieval capabilities. By following simple steps connecting, writing SQL, and managing data you can create systems such as inventory apps, school databases, employee records, and more.

5.3.4 Executing SQL queries using Python

After successfully connecting to a database, the next step is to execute SQL queries to manage and manipulate data. This includes tasks such as inserting new records, reading data, updating existing values, and deleting unwanted entries.

In Python, SQL queries are executed using a cursor object, which is created from the database connection. The cursor acts as a control point to issue SQL commands and retrieve data from the database. Python's built-in or third-party database connectors (such as `sqlite3` or `mysql.connector`) provide the required functions to perform these operations. The general syntax for executing SQL queries using Python is:

# Step 1: Create a cursor object

cursor = connection.cursor()

# Step 2: Execute an SQL query

cursor.execute(sql_query, parameters)

# Step 3: Commit changes (if data is modified)

connection.commit() # Only for INSERT, UPDATE, DELETE

# Step 4: Fetch results (for SELECT queries)

rows = cursor.fetchall() # or cursor.fetchone()

Here,

- `cursor()` – Creates a cursor to execute SQL statements.
- `execute()` – Executes an SQL command; parameters can be passed securely.
- `commit()` – Saves changes to the database (not required for SELECT).
- `fetchall()` – Retrieves all results from a SELECT query.
- `fetchone()` – Retrieves only the first result from a SELECT query.

5.3.4.1 SELECT- Retrieving Data from the Table

The `SELECT` statement in SQL is used to retrieve data from a table. In Python, this is done using the `cursor.execute()` function followed by a query such as `SELECT * FROM table_name`. The `*` symbol means "all columns," so the query will return every column and row from the specified table. After executing the query, you use methods like `fetchall()` to retrieve all the matching rows or `fetchone()` to get just a single row. The result is typically returned as a list of tuples, where each tuple represents a row in the table.

Syntax:

```
cursor.execute("SELECT * FROM table_name")

result = cursor.fetchall()
```

Example: Retrieving All Students from the Table

Assume we have a table named `students` with columns `id`, `name`, and `age`.

```
import mysql.connector

# Step 1: Connect to the database

conn = mysql.connector.connect(

    host="localhost",

    user="root",

    password="your_password",

    database="school"

)

# Step 2: Create cursor object

cursor = conn.cursor()
```

```
# Step 3: Execute SELECT query

cursor.execute("SELECT * FROM students")

# Step 4: Fetch all rows

result = cursor.fetchall()

# Step 5: Print each row

for row in result:

    print(row)

# Step 6: Close the connection

cursor.close()

conn.close()
```

To retrieve data from a MySQL database in Python, you first connect to the database and create a cursor object. Then, using the `cursor.execute()` function, you run a `SELECT` query. The returned result can be stored in a variable and looped through using a `for` loop to display or process each row. For example, retrieving all students from a `students` table can be done with the query `SELECT * FROM students`, and the output can be printed row by row. This operation is read-only, so you don't need to call `commit()` after a `SELECT` query. Retrieving data is a fundamental part of database interaction, allowing Python programs to access and work with stored information in a dynamic way.

5.3.4.2 INSERT- Adding Data to the Table

The `INSERT` statement in SQL is used to add new data into a table. When working with Python and MySQL, this operation is carried out using the `cursor.execute()` function along with an `INSERT INTO` SQL query. The `INSERT` command specifies the table name and the columns where data should be placed, followed by the `VALUES` clause that holds the data to be inserted. In Python, it is a best practice to use parameterized queries with placeholders (`%s`) instead of hardcoding the values directly into the SQL string. This approach helps protect the database from SQL injection attacks and ensures safe handling of user input.

Syntax:

```
cursor.execute("INSERT INTO table_name (col1, col2) VALUES (%s, %s)", (val1,
val2))
```

Example: Add new student

Let's add a new student to the `students` table with the fields: `name` and `age`.

```
import mysql.connector

# Step 1: Connect to the database
```

```
conn = mysql.connector.connect(
    host=”localhost”,
    user=”root”,
    password=”your_password”,
    database=”school”
)
# Step 2: Create a cursor
cursor = conn.cursor()
# Step 3: Prepare the INSERT statement
sql = “INSERT INTO students (name, age) VALUES (%s, %s)”
val = (“Alice”, 20)
# Step 4: Execute the statement
cursor.execute(sql, val)
# Step 5: Commit the transaction
conn.commit()
# Step 6: Confirmation message
print(“1 record inserted successfully.”)
```

To perform an insert operation, you first create a connection to the database using `mysql.connector.connect()`, then create a cursor object using `conn.cursor()`. You then define the SQL insert query and provide the values in a tuple. The `cursor.execute()` function runs the command, and `conn.commit()` is called to save the changes permanently to the database. If `commit()` is not called, the inserted data will not be saved. This process is essential for adding new rows into tables such as adding a new student record with their name and age into a `students` table. After execution, you can optionally print a confirmation message or run a `SELECT` query to verify that the data has been added successfully.

5.3.4.3 UPDATE- Modifying Existing Data

The `UPDATE` statement in SQL is used to **modify existing records** in a table. In Python, this is achieved by executing an `UPDATE` SQL command using the `cursor.execute()` function. You can change one or more columns for selected rows by specifying a `WHERE` condition to target the desired records. If you omit the `WHERE` clause, all rows in the table will be updated - so it’s important to use it carefully.

Syntax:

```
sql = "UPDATE table_name SET column1 = %s WHERE condition_column = %s"

val = (value1, condition_value)

cursor.execute(sql, val)

connection.commit()
```

Here,

- `cursor.execute()` runs the SQL update command.
- `commit()` is necessary to save the changes to the database.
- `%s` is used as a placeholder for values to prevent SQL injection.

Example: Updating a Student's Age

Suppose we want to update the age of a student named "Alice" from 20 to 21 in the `students` table.

```
import mysql.connector
# Step 1: Connect to the database
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="your_password",
    database="school"
)
# Step 2: Create a cursor object
cursor = conn.cursor()
# Step 3: Prepare the SQL update query
sql = "UPDATE students SET age = %s WHERE name = %s"
val = (21, "Alice")
# Step 4: Execute the update query
cursor.execute(sql, val)
# Step 5: Commit the changes
conn.commit()
# Step 6: Confirmation message
```

print(cursor.rowcount, “record(s) updated.”)

# Step 7: Close the connection

cursor.close()

conn.close()

In this example, we update the age of a student named “Alice” in the `students` table. First, we establish a connection to the MySQL database using the `mysql.connector.connect()` function and create a cursor object to execute SQL commands. The `UPDATE` query is prepared using placeholders `%s` for the values to be substituted, which helps prevent SQL injection. The SQL command `"UPDATE students SET age = %s WHERE name = %s"` is designed to change the `age` column to `21` only for the student whose `name` is `"Alice"`. These values are passed as a tuple `(21, "Alice")` to the `cursor.execute()` function. After executing the query, `conn.commit()` is called to apply the changes permanently in the database. Finally, the program prints the number of rows affected using `cursor.rowcount`, which confirms that one record was successfully updated. This operation demonstrates how to safely and efficiently modify data in a MySQL database using Python.

5.3.4.4 DELETE- Removing Data from the Table

The `DELETE` statement in SQL is used to remove one or more records from a table. In Python, you execute a `DELETE` query using the cursor object. It is very important to include a `WHERE` clause in your `DELETE` statement to specify which rows to remove; otherwise, all rows in the table will be deleted, which can lead to data loss.

Syntax:

sql = “DELETE FROM table_name WHERE column_name = %s”

val = (value,)

cursor.execute(sql, val)

connection.commit()

Here,

- The `%s` is a placeholder for the value used in the condition.
- The value should be provided as a tuple, even if there is only one item.
- Always call `commit()` to save the changes.

Example: Deleting a Student Record

Suppose we want to delete the record of a student named “Alice” from the `students` table.

import mysql.connector

# Step 1: Connect to the database

```
conn = mysql.connector.connect(
    host=”localhost”,
    user=”root”,
    password=”your_password”,
    database=”school”
)
# Step 2: Create cursor
cursor = conn.cursor()
# Step 3: Prepare the DELETE query
sql = “DELETE FROM students WHERE name = %s”
val = (“Alice”,)
# Step 4: Execute the query
cursor.execute(sql, val)
# Step 5: Commit the transaction
conn.commit()
# Step 6: Confirmation message
print(cursor.rowcount, “record(s) deleted.”)
# Step 7: Close the connection
cursor.close()
conn.close()
```

In this example, the program deletes the record of the student named “Alice” from the `students` table. After connecting to the database and creating a cursor object, the SQL `DELETE` statement is prepared with a placeholder `%s` to specify the name. The value `(“Alice”)` is passed as a tuple to safely fill the placeholder. The `cursor.execute()` method runs the delete command, and `conn.commit()` saves the changes to the database. Finally, the program prints how many records were deleted using `cursor.rowcount`. Including the `WHERE` clause ensures only the targeted record is removed, preventing accidental deletion of the entire table.

Recap

- A **database** is a structured collection of data stored electronically for easy access and management.
- **SQL (Structured Query Language)** is the standard language used to communicate with relational databases.
- Python can interact with databases using modules such as **`sqlite3`** and **`mysql.connector`**.
- To work with a database in Python, you first establish a **connection** using the appropriate connector module.
- The **cursor** object is used to execute SQL commands and fetch results.
- The **`SELECT`** statement retrieves data from a database table.
- Using **`SELECT *`** fetches all columns and rows from a table.
- Data retrieved by `SELECT` queries can be fetched using **`fetchall()`** or **`fetchone()`** methods.
- The **`INSERT INTO`** statement adds new records to a table.
- It is important to use **parameterized queries** (using `%s` placeholders) to prevent SQL injection.
- After executing `INSERT`, `UPDATE`, or `DELETE` queries, always call **`commit()`** to save changes.
- The **`UPDATE`** statement modifies existing records based on specified conditions.
- The **`WHERE`** clause in `UPDATE` and `DELETE` statements restricts the operation to specific rows.
- The **`DELETE FROM`** statement removes records from a table.
- Omitting the `WHERE` clause in `UPDATE` or `DELETE` will affect all rows in the table.
- The **`cursor.execute()`** method runs an SQL query from Python.
- The connection should be closed using **`conn.close()`** after database operations are complete.
- Exception handling is recommended to manage database errors and ensure clean connection closure.
- Working with databases enables Python programs to become dynamic and data-driven.
- Mastery of basic SQL commands (SELECT, INSERT, UPDATE, DELETE) is essential for effective database programming in Python.

Objective Type Questions

1. What is a database?
2. Which language is primarily used to interact with relational databases?
3. Which Python module is used to connect with SQLite databases?
4. What function is used to establish a connection to a MySQL database in Python?
5. What is the purpose of a cursor in database programming?
6. Which SQL command is used to retrieve data from a table?
7. What does the SQL command `INSERT INTO` do?
8. Why should parameterized queries be used in Python SQL operations?
9. Which Python method saves changes made to the database?
10. What SQL statement is used to change existing data in a table?
11. How do you delete records from a table using SQL?
12. What does `fetchall()` do in Python database programming?
13. What happens if you omit the `WHERE` clause in an `UPDATE` statement?
14. How do you safely pass values to an SQL query in Python?
15. What Python statement is used to close a database connection?
16. Which SQL keyword retrieves all columns from a table?
17. In Python, which method fetches only the first row of a query result?
18. What is the first step in interacting with a database in Python?
19. What Python module can be used to connect to MySQL databases?
20. Why is `commit()` important after executing INSERT, UPDATE, or DELETE?

Answers to Objective Type Questions

1. A structured collection of data stored electronically.
2. SQL (Structured Query Language).
3. `sqlite3`.

4. `mysql.connector.connect()`.
5. To execute SQL queries and fetch data from the database.
6. `SELECT`.
7. Adds new records into a table.
8. To prevent SQL injection attacks and ensure safe data handling.
9. `connection.commit()`.
10. `UPDATE`.
11. Using the `DELETE FROM` statement with a `WHERE` clause.
12. Retrieves all rows of a query result.
13. All records in the table will be updated.
14. Using placeholders like `%s` and passing values as a tuple.
15. `connection.close()`.
16. `SELECT *`.
17. `fetchone()`.
18. Establishing a connection to the database.
19. `mysql.connector`.
20. It saves the changes permanently to the database.

Assignments

1. Explain the process of connecting Python to a MySQL database. Write a Python program that connects to a MySQL database named `school`.

2. Write a Python program to insert three new student records into a table named `students` with columns `id`, `name`, and `age`. Make sure to use parameterized queries.

3. Using Python, write a program to retrieve and display all records from the `students` table. Explain how the data fetched by the program can be processed.

4. Write a Python script to update the age of a student in the `students` table based on their name. Also, write a program to delete a student record based on the student's name.

Reference

1. Beighley, L., & Morrison, M. (2017). *Head First SQL: Your Brain on SQL – A Learner's Guide* (1st ed.). O'Reilly Media.

2. Sweigart, A. (2019). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners* (2nd ed.). No Starch Press.

3. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.

4. Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python* (2nd ed.). O'Reilly Media.

5. Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming* (1st ed.). O'Reilly Media.

Suggested Reading

1. W3Schools. (n.d.). *SQL Tutorial*. https://www.w3schools.com/sql/

2. Real Python. (2021). *Working With Databases in Python*. https://realpython.com/python-database-sqlite/

3. MySQL. (n.d.). *MySQL Connector/Python Developer Guide*. https://dev.mysql.com/doc/connector-python/en/

4. TutorialsPoint. (n.d.). *Python MySQL Tutorial*. https://www.tutorialspoint.com/python/python_mysql.html

5. GeeksforGeeks. (2022). *Python MySQL Database Access*. https://www.geeksforgeeks.org/python-mysql-database-access/

Unit 4
Applications of Database Programming in Python

Learning Outcomes

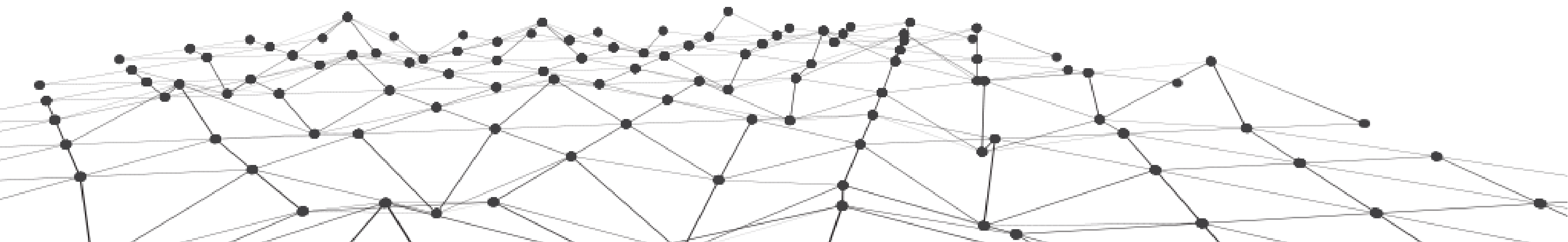
After the successful completion of the course, the learner will be able to:

- recall the basic SQL commands (SELECT, INSERT, UPDATE, DELETE) used in CRUD operations with Python's sqlite3 module.
- explain how menu-driven applications simplify database interaction using command-line or GUI interfaces.
- develop a Python application using sqlite3 to perform basic CRUD operations
- differentiate between console-based and GUI-based database applications in Python
- examine the structure and flow of a Python program that integrates Tkinter with SQLite to manage forms and generate reports.

Prerequisites

Managing information is a key part of any organization. For example, a college needs to keep records of many students, including their names, courses, and grades. When a new student joins or updates are needed, the system must handle the data quickly and correctly. Database programming helps in doing this. Using Python with a database like SQLite allows the creation of applications that can add, view, update, or delete data. These are called CRUD operations. These tasks can be done through simple menus or graphical windows using Tkinter. This topic is important because it connects basic coding with real-world data use, helping in building useful applications.

Some basic knowledge is needed to learn this topic. It is important to understand Python programming such as variables, loops, and functions. Knowing database terms like tables, rows, and simple SQL commands (SELECT, INSERT, UPDATE, DELETE) is also helpful. A basic idea of object-oriented programming and Tkinter makes it easier to create visual applications. Learning this topic builds skills to make systems like student records or inventory tools. It also improves thinking and coding ability and prepares learners to build larger software projects in the future.



Key words

GUI Programming, Tkinter, Forms, Reports, Treeview, Data Storage, Data Retrieval, Database Programming, SQLite, CRUD Operations

Discussion

5.4.1 Applications of Database Programming

Database programming plays an essential role in building robust, efficient, and secure data-driven systems. It involves integrating programming languages like Python with databases to facilitate effective data management, querying, storage, and processing. Python provides versatile modules such as sqlite3, mysql-connector-python, and SQLAlchemy, making it a preferred language for database development across different sectors.

5.4.2 Creating Menu-Driven Database Applications

Menu-driven applications offer a user-friendly way to interact with a database through a series of options. These options allow users to perform operations like adding, viewing, editing, or deleting records without directly interacting with SQL queries. Creating menu-driven database applications in SQLite using Python involves designing an interactive program that allows users to perform database operations such as Create, Read, Update, and Delete (CRUD) through a command-line or GUI menu interface. Below are the step-by-step guidelines to build such an application using the built-in sqlite3 module in Python.

Step 1: Import sqlite3 Module

```
import sqlite3
```

Step 2: Connect to Database

```
conn = sqlite3.connect("student.db")

cursor = conn.cursor()
```

Step 3: Create a Table (If Not Exists)

```
cursor.execute("""

CREATE TABLE IF NOT EXISTS student (

    id INTEGER PRIMARY KEY AUTOINCREMENT,

    name TEXT NOT NULL,

    age INTEGER,

    grade TEXT
)
""")
conn.commit()
```

Step 4: **Define CRUD Operation Functions**

```
// Insert Function
def insert_student(name, age, grade):
    cursor.execute("INSERT INTO student (name, age, grade) VALUES (?, ?, ?)",
(name, age, grade))
    conn.commit()
    print("Student added successfully!")
//Display All Records
def display_students():
    cursor.execute("SELECT * FROM student")
    records = cursor.fetchall()
    for row in records:
        print(row)
// Search by ID
def search_student(student_id):
    cursor.execute("SELECT * FROM student WHERE id=?", (student_id,))
    record = cursor.fetchone()
    if record:
        print(record)
    else:
        print("Student not found.")
//Update a Record
def update_student(student_id, name, age, grade):
    cursor.execute("UPDATE student SET name=?, age=?, grade=? WHERE id=?",
(name, age, grade, student_id))
```

```
    conn.commit()
    print("Student record updated.")
// Delete a Record
def delete_student(student_id):
    cursor.execute("DELETE FROM student WHERE id=?", (student_id,))
    conn.commit()
    print("Student record deleted.")
```

Step 5: **Create the Menu**

```
def menu():
    while True:
        print("\n--- Student Database Menu ---")
        print("1. Add Student")
        print("2. Display All Students")
        print("3. Search Student by ID")
        print("4. Update Student")
        print("5. Delete Student")
        print("6. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            name = input("Enter name: ")
            age = int(input("Enter age: "))
            grade = input("Enter grade: ")
            insert_student(name, age, grade)
        elif choice == '2':
            display_students()
        elif choice == '3':
            student_id = int(input("Enter student ID to search: "))
        search_student(student_id)
    elif choice == ‘4’:
        student_id = int(input(“Enter student ID to update: “))
        name = input(“Enter new name: “)
        age = int(input(“Enter new age: “))
        grade = input(“Enter new grade: “)
        update_student(student_id, name, age, grade)
    elif choice == ‘5’:
        student_id = int(input(“Enter student ID to delete: “))
        delete_student(student_id)
    elif choice == ‘6’:
        print(“Exiting program.”)
        break
    else:
        print(“Invalid choice. Please try again.”)
```

Step 6: Call the Menu Function

```
menu()
```

Step 7: **Close the Connection**

```
conn.close()
```

Output

```
--- Student Database Menu ---
1. Add Student
2. Display All Students
3. Search Student by ID
4. Update Student
5. Delete Student
6. Exit
```

```
Enter your choice: 1

Enter name: Alice

Enter age: 20

Enter grade: A

Student added successfully!
```

Example: A Simple Inventory Management System using SQLite

```
import sqlite3

conn = sqlite3.connect('inventory.db')

cursor = conn.cursor()

def create_table():

    """Initializes the database table if it doesn't exist."""

    cursor.execute('''

        CREATE TABLE IF NOT EXISTS products (

            id INTEGER PRIMARY KEY AUTOINCREMENT,

            name TEXT NOT NULL,

            quantity INTEGER NOT NULL,

            price REAL NOT NULL

        )

    ''')

    conn.commit()

def add_product():

    """Adds a new product to the database."""

    name = input("Enter product name: ")

    quantity = int(input("Enter quantity: "))

    price = float(input("Enter price: "))

    cursor.execute("INSERT INTO products (name, quantity, price) VALUES (?, ?, 
?)", (name, quantity, price))
```

```
    conn.commit()
    print("Product added successfully.")
def view_products():
    """"Displays all products in the database.""""
    cursor.execute("SELECT * FROM products")
    products = cursor.fetchall()
    if not products:
        print("No products in the inventory.")
        return
    print("\n--- Current Inventory ---")
    for product in products:
   print(f'ID: {product[0]}, Name: {product[1]}, Quantity: {product[2]}, Price:
${product[3]:.2f}")
    print("-------------------------")
def main_menu():
    """"The main loop for the menu-driven application.""""
    create_table()  # Ensure the table exists
    while True:
        print("\n--- Inventory Management System ---")
        print("1. Add a new product")
        print("2. View all products")
        print("3. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            add_product()
        elif choice == '2':
            view_products()
        elif choice == '3':
            print("Exiting...")
            conn.close()
            break
        else:
            print("Invalid choice. Please try again.")
if __name__ == "__main__":
    main_menu()
```

Output

```
--- Inventory Management System ---
1. Add a new product
2. View all products
3. Exit
Product added successfully.
--- Inventory Management System ---
1. Add a new product
2. View all products
3. Exit
--- Current Inventory ---
ID: 1, Name: book, Quantity: 10, Price: $300.00
-------------------------
--- Inventory Management System ---
1. Add a new product
2. View all products
3. Exit
Invalid choice. Please try again.
--- Inventory Management System ---
```

```
1. Add a new product

2. View all products

3. Exit

Exiting...
```

5.4.3 Designing simple forms and reports in python

Designing **simple forms and reports in Python** involves collecting user input (forms) and presenting structured data output (reports). Below is a structured approach to create **console-based forms and reports**, which can be further enhanced with GUI (e.g., Tkinter) or web frameworks (e.g., Flask).

```
# Tkinter-based Python application- A form to input employee data A report
window to display stored employee data from SQLite database

import sqlite3

from tkinter import *

from tkinter import messagebox, ttk

# Database Setup

conn = sqlite3.connect("employee.db")

cursor = conn.cursor()

cursor.execute("""

CREATE TABLE IF NOT EXISTS employee (

    id INTEGER PRIMARY KEY AUTOINCREMENT,

    name TEXT NOT NULL,

    department TEXT,

    salary REAL

)

""")

conn.commit()

#  Save Employee Function

def save_employee():

    name = name_entry.get()
    dept = dept_entry.get()
    try:
        salary = float(salary_entry.get())
    except ValueError:
        messagebox.showerror("Input Error", "Please enter a valid salary")
        return
    if name and dept:
   cursor.execute("INSERT INTO employee (name, department, salary) VALUES
(?, ?, ?)",
                    (name, dept, salary))
        conn.commit()
        messagebox.showinfo("Success", "Record Saved!")
        name_entry.delete(0, END)
        dept_entry.delete(0, END)
        salary_entry.delete(0, END)
    else:
        messagebox.showwarning("Missing Info", "Please fill all fields")
```

# View Report Function

```
def show_report():
    report_win = Toplevel(root)
    report_win.title("Employee Report")
    report_win.geometry("500x300")
 tree = ttk.Treeview(report_win, columns=("ID", "Name", "Department",
"Salary"), show='headings')
    tree.heading("ID", text="ID")
    tree.heading("Name", text="Name")
    tree.heading("Department", text="Department")
    tree.heading("Salary", text="Salary")
    cursor.execute(“SELECT * FROM employee”)
    rows = cursor.fetchall()
    for row in rows:
    tree.insert(“”, END, values=row)
    tree.pack(fill=BOTH, expand=True)
```

# Main Window

```
root = Tk()
root.title(“Employee Entry Form”)
root.geometry(“400x300”)
```

# Form Labels and Entries

```
Label(root, text=”Employee Name:”).pack(pady=5)
name_entry = Entry(root, width=30)
name_entry.pack()
Label(root, text=”Department:”).pack(pady=5)
dept_entry = Entry(root, width=30)
dept_entry.pack()
Label(root, text=”Salary:”).pack(pady=5)
salary_entry = Entry(root, width=30)
salary_entry.pack()
```

# Buttons

```
Button(root, text=”Save Record”, command=save_employee, bg=”green”,
fg=”white”).pack(pady=10)
Button(root, text=”Show Report”, command=show_report, bg=”blue”,
fg=”white”).pack(pady=5)
root.mainloop()
```

# Close database when GUI ends

```
conn.close()
```

5.4.4 Implementing CRUD operations in real time scenarios

CRUD stands for **Create, Read, Update, and Delete**, the four basic functions of persistent storage. These operations form the foundation of **database programming** in real-time applications such as web apps, desktop apps, mobile apps, and enterprise systems. Let's consider a **Student Management System** for a college. The following system is an example of real time scenario:

- **Create** new student records.
- **Read** (view) student details.
- **Update** existing student information.
- **Delete** records when a student leaves or is no longer valid.

Setup:

```
import sqlite3

# Connect to SQLite database (creates a new one if it doesn’t exist)

conn = sqlite3.connect(‘college.db’)

cursor = conn.cursor()

# Create student table

cursor.execute(‘”

CREATE TABLE IF NOT EXISTS students (

   id INTEGER PRIMARY KEY AUTOINCREMENT,

   name TEXT NOT NULL,

   age INTEGER,

   course TEXT

)

‘”)

conn.commit()
```

CREATE Operation (Insert a New Student)

```
def create_student(name, age, course):

   cursor.execute(“INSERT INTO students (name, age, course) VALUES (?, ?, ?)”, (name, age, course))

   conn.commit()

   print(“Student record created.”)
```

```
# Example usage
create_student("Anjali", 21, "MCA")
```

READ Operation (View All Students)

```
def read_students():
    cursor.execute("SELECT * FROM students")
    records = cursor.fetchall()
    for row in records:
        print(row)
# Example usage
read_students()
```

UPDATE Operation (Modify Student Details)

```
def update_student(student_id, new_course):
    cursor.execute("UPDATE students SET course = ? WHERE id = ?", (new_course, student_id))
    conn.commit()
    print("Student record updated.")
# Example usage
update_student(1, "Data Science")
```

DELETE Operation (Remove Student Record)

```
def delete_student(student_id):
    cursor.execute("DELETE FROM students WHERE id = ?", (student_id,))
    conn.commit()
    print("Student record deleted.")
# Example usage
delete_student(1)
```

Recap

- Database programming involves integrating Python with databases for efficient data storage, querying, and management.
- Python provides useful modules for database development such as sqlite3, mysql-connector-python, and SQLAlchemy.
- It helps build robust, secure, and scalable data-driven applications used in various domains.
- Menu-driven applications provide a user-friendly interface to perform database operations without writing SQL queries.
- A menu-driven student database application can be created using Python and SQLite with CRUD functionalities.
- The application allows users to add, view, search, update, and delete student records through a simple text menu.
- Each database operation is written as a separate Python function using SQL commands.
- A loop-based menu system lets users repeatedly choose actions until they exit the program.
- Designing forms and reports in Python involves taking user inputs and displaying stored data in a readable format.
- Tkinter is used to design a simple GUI form to input employee details such as name, department, and salary.
- The GUI also includes a report window that displays all saved records using a Treeview table layout.
- CRUD stands for **Create, Read, Update, and Delete**, which are the four basic operations for managing data in a database.
- These operations are essential for building **interactive and dynamic applications** such as web apps, desktop software, and mobile apps.
- Python with **SQLite** is commonly used to perform CRUD operations efficiently in real-time systems.
- Create operation allows inserting new records into the database (e.g., adding a new student in a college system).
- Read operation retrieves and displays data from the database (e.g., viewing student details or product list).

- ♦ Update operation modifies existing records (e.g., changing a student’s course or updating inventory quantity).
- ♦ Delete operation removes records that are no longer needed (e.g., deleting a student record when they leave).
- ♦ Each operation is implemented using SQL commands (INSERT, SELECT, UPDATE, DELETE) in Python functions.
- ♦ These operations ensure data integrity, user interaction, and dynamic content management in applications.

Objective Type Questions

1. Which module in Python is commonly used to interact with SQLite databases?
2. What does the acronym CRUD stand for?
3. Which SQL command is used to retrieve data from a table?
4. Which function in Python is used to execute SQL statements?
5. What keyword is used in SQL to add new records to a table?
6. What Python method is used to save changes to a SQLite database?
7. Which method is used to close the database connection in Python?
8. What SQL command is used to modify existing records?
9. What SQL command is used to remove records from a database?
10. Which clause is used in SQL to specify conditions for selection or modification?
11. What keyword is used to create a table only if it does not exist?
12. Which Python module is used to create GUI forms in desktop applications?
13. Which data structure is used in Tkinter to display tabular data as reports?
14. What is the extension of an SQLite database file created by Python?

Answers to Objective Type Questions

1. sqlite3
2. Create
3. SELECT
4. execute
5. INSERT
6. commit
7. close
8. UPDATE
9. DELETE
10. WHERE
11. IF NOT EXISTS
12. Tkinter
13. Treeview
14. .db

Assignments

1. Explain the concept of CRUD operations in database programming with suitable examples.
2. Describe the step-by-step process of creating a menu-driven database application in Python using SQLite.
3. Discuss the role of the sqlite3 module in Python. How does it support database programming?
4. Explain the process of designing and displaying database reports in a Python GUI application

Reference

1. Grinberg, M. (2023). *Flask web development: Developing web applications with Python* (2nd ed.). O'Reilly Media.

2. Zelle, J. M. (2022). *Python programming: An introduction to computer science* (4th ed.). Franklin, Beedle & Associates.

3. Slatkin, B. (2020). *Effective Python: 90 specific ways to write better Python* (2nd ed.). Addison-Wesley Professional.

4. Stewart, R. (2021). *Python GUI Programming with Tkinter: Design and build functional and user-friendly GUI applications*. Packt Publishing.

5. Owain, G. (2023). *Mastering SQLite for Python Developers*. Independently published.

Suggested Reading

1. Pillai, B. (2022). *Python GUI development with Tkinter: Design responsive and robust GUI applications*. Independently published.

2. **Lutz, M. (2021).** *Learning Python* (5th ed.). O'Reilly Media.

3. Sarkar, D. (2021). *Python GUI programming with Tkinter and SQLite*. BPB Publications.

4. Zelle, J. M. (2017). Python Programming: An Introduction to Computer Science (3rd ed.). Franklin, Beedle & Associates.

5. Downey, A. (2015). Think Python: How to Think Like a Computer Scientist (2nd ed.). O'Reilly Media.

6. Python Official Documentation. (n.d.). Errors and Exceptions – Python 3.x Docs. https://docs.python.org/3/tutorial/errors.html

6

BLOCK

Familiarizing NumPy, Matplotlib and Pandas

Unit 1
Familiarizing NumPy

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- define NumPy and its role in Python programming.
- list the methods to create NumPy arrays from existing data and built-in functions.
- describe array operations like indexing, slicing, reshaping, and sorting.
- explain mathematical and statistical functions used in NumPy.

Prerequisites

If you have already learned how to use lists and loops in Python, you have taken the first step toward working with data in programming. You might have written small programs to store numbers in a list and used loops to add or print them. While this works well for simple tasks, it can become slow and complicated when the data gets larger or more complex. This is where NumPy becomes useful. NumPy builds on what you already know about Python lists, but it helps you do morefaster and more efficiently. In this unit, we will learn how to use NumPy arrays to perform mathematical operations easily, handle large datasets, and write cleaner programs. This knowledge will help you move from basic programming to more advanced data analysis and scientific computing.

Key words

NumPy, arrays, indexing, slicing, statistical operations

Discussion

6.1.1 What is NumPy?

NumPy stands for Numerical Python. It is a tool (called a library) used in Python programming to work with numbers and data more efficiently. It helps us perform mathematical and scientific calculations quickly and easily. The main feature of NumPy is something called an array, which helps to store and work with many numbers easily. An array is just a special kind of list where we can store many numbers together.

6.1.1.1 Why is NumPy Important?

1. Faster Calculations

Normally, when we use loops (like for loops) in Python to do math on many numbers, it can be slow. NumPy uses a technique called vectorization that allows it to do all the calculations much faster, without writing loops.

2. Used in Many Other Tools

NumPy is not just useful by itself, many popular Python tools like Pandas (for data tables), Matplotlib (for graphs), SciPy (for scientific computing), and TensorFlow (for machine learning) are built on top of NumPy. So, learning NumPy helps you use all these tools better.

3. Good for Big Data

When we deal with a lot of data, like thousands or millions of numbers, NumPy can handle it efficiently. It saves memory and time, which is very important in areas like research, finance, and engineering.

4. No Need for Loops

In regular Python, you often write loops to go through lists and do calculations. With NumPy, you can perform the same task in just one line, making your code cleaner and faster.

6.1.2 Arrays in NumPy

Arrays in NumPy are the foundation of numerical computing in Python. Unlike normal Python lists, NumPy arrays (called ndarrays) have special features like fixed size, the ability to handle multiple dimensions, and support for many fast and memory-efficient mathematical operations. They allow you to perform calculations on entire datasets at once, without using loops. This makes NumPy a key tool for tasks in data analysis, machine learning, scientific research, and engineering.

6.1.2.1 Array attributes

NumPy arrays, also known as ndarray objects, include several built-in attributes that allow users to easily inspect and work with their structure and data. These attributes provide important information about the array, such as its shape, size, and number of dimensions. One of the most commonly used attributes is:

1. Number of Dimensions (.ndim)

This attribute returns the number of axes (dimensions) in a NumPy array. For example, a 1D array has one dimension, a 2D array (like a matrix) has two dimensions, and so on.

Example:

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.ndim)
```

Output:

```
2
```

The array a has two lists inside it, each with three elements. So, it is a 2-dimensional array (2D), and .ndim returns 2.

2. Dimensions of the Array (.shape)

The .shape attribute returns a tuple that shows the size of the array in each dimension. It tells us how many elements are present along each axis such as the number of rows and columns in a 2D array.

Example:

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.shape)
```

Output:

```
(2, 3)
```

The array a has 2 rows and 3 columns, so the shape of the array is represented as the tuple (2, 3).

3. Total Number of Elements (.size)

The .size attribute gives the total number of elements stored in the array. It is calculated by multiplying the size of each dimension.

Example:

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.size)
```

Output:

```
6
```

There are 2 rows and 3 columns, making a total of 2 × 3 = 6 elements in the array.

4. Data Type of Elements (.dtype)

The .dtype attribute displays the data type of the elements stored in a NumPy array. NumPy automatically assigns a suitable data type based on the values in the array, but it can also be specified manually when creating the array.

Example:

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.dtype)
```

Output:

```
int64
```

The array a contains integer values, so NumPy has assigned the data type as int64 (64-bit integer). Note that the exact data type may vary depending on your system architecture for example, it might show as int32 on some systems.

6.1.3 Creating and Manipulating NumPy Arrays

Before we can use any functions from the NumPy library, we need to import it into our Python program. This means we are telling Python to make the NumPy tools available for use.

To do this, we use the following line of code:

```
import numpy as np
```

In this line of code:

- The word import tells Python to bring in extra tools from outside.
- numpy is the name of the tool (library) we want to use.
- as np means we are giving NumPy a short name (np) so it's easier to use later.

Let us now look at a basic example to understand the difference between using regular Python and using NumPy. Suppose we have two lists of numbers, and we want to add the numbers in the same positions from both lists. We will see how this is done first without NumPy, and then using NumPy to show how NumPy makes it easier and faster.

Without NumPy:

```
a = [1, 2, 3]
```

```
b = [4, 5, 6]
result = []
for i in range(len(a)):
    result.append(a[i] + b[i])
print(result)
```

With NumPy:

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(a + b)
```

6.1.3.1 One-Dimensional Array

A one-dimensional array (Fig 6.1.1) in NumPy is essentially a sequence or list of elements, all of which are of the same type. It is a type of linear array. It is similar to a Python list, but offers the benefit of faster operations and a more compact memory footprint. A 1D array as a vector, where each element is accessed by its index.

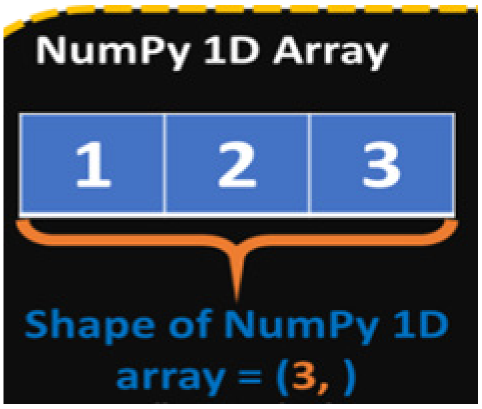


Fig 6.1.1 One Dimensional Array

Sample Program:

```
import numpy as np
arr = np.array([10, 20, 30, 40])
print("Array:", arr)
```

Ouput:

```
Array: [10 20 30 40]
```

This example demonstrates how to create a one-dimensional array using NumPy.

6.1.3.2 Two-Dimensional Array

A two-dimensional array (2D array) in NumPy is essentially a grid or matrix, where elements are arranged in rows and columns as shown in Fig 6.1.2. This structure can represent things like images, tables of data, and more complex mathematical objects. Data is stored in tabular form.

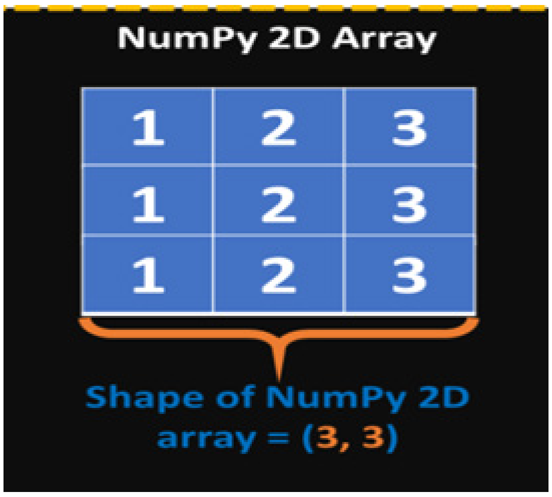


Fig 6.1.2 Two Dimensional Array

Let's see how a two-dimensional array is created and printed using NumPy.

Sample Program:

```
import numpy as np

arr2d = np.array([[1, 2, 3], [4, 5, 6]])

print("2D Array:\n", arr2d)
```

Output:

```
2D Array:

[[1 2 3]

[4 5 6]]
```

6.1.3.3 N- Dimensional Array

ndarray (short for *N-dimensional array*) is a core object in NumPy. It is a homogeneous array which means it can hold elements of the same data type. An N-dimensional array (or ndarray) in NumPy is an array with N dimensions, where N can be any non-negative integer. While 1D and 2D arrays are the most commonly used, NumPy allows you to work with arrays of higher dimensions (3D, 4D, etc.) to represent more complex data structures like tensors, volumetric data, or time-series data in multiple dimensions as shown in Fig 6.1.3.

The key advantage of using N-dimensional arrays is that they allow you to work efficiently with large datasets in various fields such as image processing, machine learning, and scientific simulations.

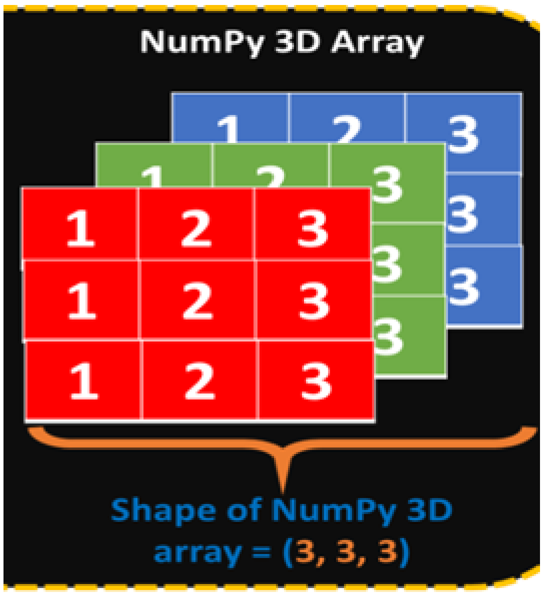


Fig 6.1.3 : N-dimensional Array

Example:

```
import numpy as np
#1D array
arr1 = np.array([1, 2, 3, 4, 5])
#2D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
#3D array
arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr1)
print(arr2)
print(arr3)
```

Ouput:

```
[1 2 3 4 5]
[[1 2 3]
[4 5 6]]
[[[1 2]
[3 4]]
[[5 6]
[7 8]]]
```

6.1.3.4 Creating NumPy Arrays from Existing Data

NumPy allows you to create arrays from existing data structures like lists, tuples, or even raw bytes. This is useful in data analysis and scientific computations. Some common methods include:

1. Using numpy.asarray()

Converts lists, tuples, or other array-like objects into NumPy arrays.

Syntax:

```
numpy.asarray(a, dtype=None, order=None)
```

- a: Input data (list, tuple, etc.)
- dtype: (Optional) Desired data type
- order: (Optional) Memory layout (C for row-major, F for column-major). Default is None, which means NumPy decides based on the input.

Example:

```
import numpy as np
list = [1, 2, 3, 4, 5]
arr_list = np.asarray(list)
print("Array from list:", arr_list)
```

Output:

```
[1 2 3 4 5]
```

2. Using numpy.frombuffer()

The numpy.frombuffer() function creates an array from a buffer object, such as bytes objects or byte arrays.

Syntax:

```
numpy.frombuffer(buffer, dtype=float, count=-1, offset=0)
```

Where

- buffer − It is the buffer object containing the data to be interpreted as an array.
- dtype (optional) − It is the desired data type of the elements in the resulting array. Default is float.

- count (optional) – It is the number of items to read from the buffer. Default is -1, which means all data is read.
- offset (optional) – It is the starting position within the buffer to begin reading data. Default is 0.

Example:

```
bytes = b'hello world'

arr_bytes = np.frombuffer(bytes, dtype='S1')

print("Array from bytes object:", arr_bytes)
```

Output:

```
[b'h' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd']
```

3. From Python Lists

Lists can be converted to arrays using np.array() or np.asarray().

Example:

```
list = [1, 2, 3, 4, 5]

arr = np.array(list)

print("Array from list:", arr)
```

Output:

```
[1 2 3 4 5]
```

4. From Python Tuples

Tuples can also be converted using np.array().

Example:

```
tuple = (1, 2, 3, 4, 5)

arr = np.array(tuple)

print("Array from tuple:", arr)
```

Output:

```
[1 2 3 4 5]
```

6.1.3.5 Creating Arrays Using Built-in Functions

NumPy provides several built-in functions to quickly create arrays without manually specifying each element. These functions are especially helpful when we want arrays filled with zeros, ones, or a sequence of numbers for testing or initialization. Let us explore four common built-in array creation functions in NumPy: zeros(), ones(), arange(), and linspace().

Sample Program:

```
import numpy as np
a = np.zeros((2, 3))
b = np.ones((2, 2))
c = np.arange(0, 10, 2)
d = np.linspace(0, 1, 5)
print("Zeros:\n", a)
print("Ones:\n", b)
print("Arange:\n", c)
print("Linspace:\n", d)
```

Output:

```
Zeros:
[[0. 0. 0.]
 [0. 0. 0.]]
Ones:
[[1. 1.]
 [1. 1.]]
Arange:
[0 2 4 6 8]
Linspace:
[0.   0.25 0.5  0.75 1.  ]
```

1. np.zeros((2, 3)):

This creates a table (or array) with 2 rows and 3 columns, and fills all the places with the number 0. It is helpful when you want to start with an empty array and add values to it later.

2. np.ones((2, 2)):

This makes a 2 by 2 table where all the values are 1. It is useful when you need an array with the same value everywhere, like in some math problems or computer programs.

3.np.arange(0, 10, 2):

This gives you a list of numbers starting from 0 and going up to (but not including) 10, with a gap of 2 between each number. So, the result will be: [0, 2, 4, 6, 8]. It's useful when you want numbers at regular steps.

4. np.linspace(0, 1, 5):

This gives you 5 numbers that are spaced equally between 0 and 1. The result will be: [0. 0.25 0.5 0.75 1.]. It helps when you want to divide a range into equal parts, like for graphs or measurements.

6.1.4 Array Operations in NumPy

Arrays are the core of NumPy, and once they are created, various operations can be performed on them. These operations make it easy to work with data efficiently and effectively.

6.1.4.1 Indexing

Indexing means accessing individual elements in the array by their position. In NumPy, each element in an array is assigned a specific index value, which represents its location.

Example: Indexing a 1D Array

```
import numpy as np

arr = np.array([5, 10, 15, 20, 25])

print("First element:", arr[0])

print("Last element:", arr[-1])
```

Output:

```
First element: 5

Last element: 25
```

6.1.4.2 Slicing

Slicing means selecting multiple elements from an array using a range of indices. Instead of accessing elements one by one, slicing allows you to extract a subset or section of an array in a concise way. This is particularly useful when you want to work with or analyze a portion of the data.

In NumPy, slicing uses the colon (:) operator to specify the start index, stop index, and an optional step size. The syntax generally follows:

array[start:stop:step]

Where,

- start: the index where the slice begins (inclusive)
- stop: the index where the slice ends (exclusive)
- step: the spacing between indices (default is 1)

Example:

```
import numpy as np

arr = np.array([0, 10, 20, 30, 40, 50, 60])

print("arr[2:5] =", arr[2:5])

print("arr[:4] =", arr[:4])

print("arr[::2] =", arr[::2])
```

Output:

```
arr[2:5] = [20 30 40]

arr[:4] = [ 0 10 20 30]

arr[::2] = [ 0 20 40 60]
```

6.1.4.3 Reshaping Arrays

Reshaping means changing the arrangement of data in rows and columns without changing the actual values. For example, 6 values in a single row can be reshaped into 2 rows and 3 columns. In NumPy, this is done using the reshape() function.

Example:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

reshaped = arr.reshape((2, 3))

print("Reshaped array:\n", reshaped)
```

Output:

```
Reshaped array:

[[1 2 3]

 [4 5 6]]
```

In this example, the shape of the array is changed from one-dimensional (1D), which contains 6 values, to a two-dimensional (2D) array with 2 rows and 3 columns. This process is called reshaping.

6.1.4.4 Sorting Arrays

Sorting means arranging the values in an array in a specific order, usually from smallest to largest (increasing order). In NumPy, you can use the sort() function to sort the elements.

Example:

```
import numpy as np
```

```
arr = np.array([30, 10, 40, 20])

sorted_arr = np.sort(arr)

print("Sorted array:", sorted_arr)
```

Output:

```
Sorted array: [10 20 30 40]
```

Sorting a 2D Array by Row

In a 2D array (like a table with rows and columns), sorting by row means arranging the values in each row in increasing order, one row at a time. NumPy's sort() function automatically sorts each row individually when used on a 2D array.

Example:

```
import numpy as np

matrix = np.array([[3, 1, 2], [9, 5, 6]])

print(np.sort(matrix))
```

Output:

```
[[1 2 3]
 [5 6 9]]
```

6.1.4.5 Mathematical and Statistical Operations

NumPy allows you to perform fast mathematical operations on whole arrays, without using loops. You can add, subtract, multiply, or divide arrays directly.

a. Example of mathematical operation:

```
import numpy as np

a = np.array([1, 2, 3])

b = np.array([4, 5, 6])

print("Addition:", a + b)

print("Multiplication:", a * b)
```

Output:

```
Addition: [5 7 9]

Multiplication: [ 4 10 18]
```

In this example, each element from array a is added to the corresponding element in array b. This means the first element of a (which is 1) is added to the first element of b (which is 4), the second element of a (2) is added to the second element of b (5), and the third element of a (3) is added to the third element of b (6). The result of this addition is a new array: [5, 7, 9]. In the same way, multiplication is also done element by element.

b. Example of statistical operation:

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

print("Mean:", np.mean(arr))

print("Median:", np.median(arr))

print("Standard Deviation:", np.std(arr))
```

Output:

```
Mean: 30.0

Median: 30.0

Standard Deviation: 14.142135
```

In this example, NumPy is used to perform basic statistical calculations on an array of numbers. The np.mean() function calculates the average of all the numbers in the array. It adds them together and divides by the total count, giving a mean of 30. The np.median() function finds the middle value in the sorted list, which is also 30 in this case. The np.std() function calculates the standard deviation, which tells us how much the values in the array vary or spread out from the mean.

Recap

- NumPy is a Python library used for working with numbers and data efficiently.
- It performs calculations faster than regular Python lists.
- The main feature of NumPy is the array, which stores multiple values of the same type.
- Arrays can be one-dimensional, two-dimensional, or multi-dimensional.
- Important attributes of arrays include the number of dimensions (ndim), shape (shape), total number of elements (size), and data type (dtype).
- Arrays can be created from lists, tuples, or using functions like zeros(), ones(), arange(), and linspace().
- Indexing and slicing help in accessing and selecting parts of an array.
- Arrays can be reshaped to change their structure and sorted to arrange values in order.
- Mathematical operations such as addition and multiplication can be performed directly on arrays.

- ♦ Statistical functions like mean, median, and standard deviation are used to analyze array data.

Objective Type Questions

1. What does NumPy stand for?
2. What is the core data structure used in NumPy?
3. Which NumPy attribute returns the number of dimensions of an array?
4. Which attribute gives the total number of elements in a NumPy array?
5. Which attribute shows the shape of an array in terms of rows and columns?
6. What attribute is used to check the data type of elements in a NumPy array?
7. Which NumPy function is used to create an array filled with zeros?
8. What function creates an array with a range of numbers and a given step size?
9. Which function returns evenly spaced numbers over a range?
10. What operation is used to access a specific element in an array by its position?
11. What operation selects a portion or subsection of an array?
12. What NumPy function is used to change the shape of an array?
13. What operation arranges the elements of an array in ascending order?

Answers to Objective Type Questions

1. Numerical Python
2. Array
3. ndim
4. size
5. shape
6. dtype
7. zeros

8. arange
9. linspace
10. Indexing
11. Slicing
12. reshape
13. sort

Assignments

1. Explain the features and importance of NumPy in Python programming.
2. Write a Python program to create a one-dimensional NumPy array and display its attributes like shape, size, and data type. Explain the output.
3. What is array slicing in NumPy? Explain with a suitable Python example.
4. Write a Python program using NumPy to find the mean, median, and standard deviation of a given array. Explain the result.
5. Describe the different ways to create NumPy arrays using functions like zeros(), ones(), and arange(). Provide one example for each.

Reference

1. https://www.w3schools.com/python/numpy/default.asp
2. *https://www.geeksforgeeks.org/numpy/python-numpy/*

Suggested Reading

1. Brown, Martin C. *Python: The complete reference*. Osborne/McGraw-Hill, 2001.
2. Jose, Jeeva. *Taming Python by Programming*. KHANNA PUBLISHING HOUSE.
3. Lutz, Mark. *Learning python: Powerful object-oriented programming*. " O'Reilly Media, Inc.", 2013.

Unit 2

Data Analysis with Pandas

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- define what Pandas is and describe its primary purpose in data analysis.
- list the main data structures in Pandas
- identify key characteristics of a Pandas Series and a DataFrame.
- recall common methods for handling missing data in Pandas
- familiarise several advantages of using Pandas for data manipulation and analysis.

Prerequisites

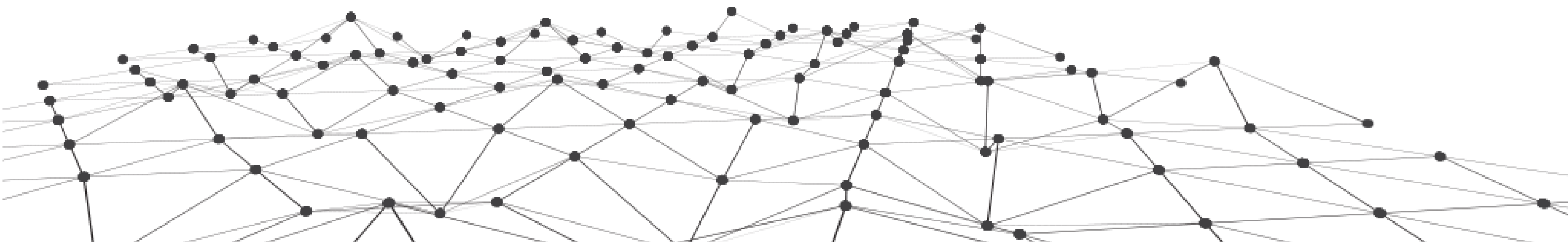
Imagine you are a teacher who needs to keep track of your students' exam scores in multiple subjects. You have a list of students with their scores in Math, English, and Science. The data is messy: some scores are missing because students missed exams, and some records are incomplete. You want to organize this data so you can quickly find a student's scores, calculate averages, and identify who needs extra help.

How can you efficiently handle, clean, and analyze this data without manually sorting through spreadsheets?

This is where **Pandas**, a powerful Python library, comes in handy. Pandas provides easy-to-use data structures like **Series** and **DataFrames** that help you organize, clean, and analyze your students' scores quickly and effectively. With Pandas, you can fill missing scores, filter students by performance, and merge new data—all with just a few lines of code!

Key words

Series, DataFrame, Dictionary, List, Index, Missing Data, Data Cleaning



Discussion

6.2.1 Introduction to Pandas

Pandas is a widely used package in Python that simplifies data analysis and manipulation. It offers intuitive and efficient ways to handle, process, and transform data. With its powerful data structures, such as Series and DataFrames, Pandas allows users to easily create, manage, and perform a variety of operations on datasets. This makes it a valuable tool for data wrangling and analytical tasks.

Pandas efficiently manages missing data and offers versatile data structures like Series for one-dimensional data and DataFrame for handling multi-dimensional datasets. It allows for quick and effective slicing of data and supports flexible operations such as merging, concatenating, and reshaping, making it a powerful tool for structured data manipulation.

Pandas works with three main data structures

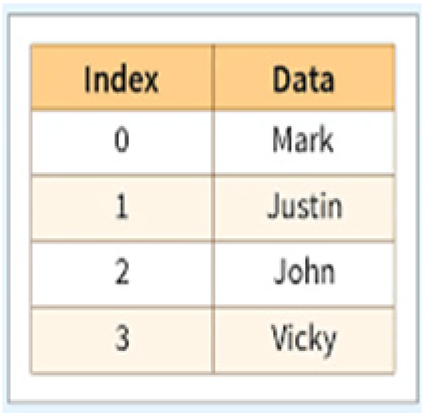
1. Series
2. DataFrame

6.2.2 Series

A **Series** is a one-dimensional array-like structure designed for handling and manipulating data. What sets it apart is its powerful and flexible **index**, which allows for efficient data access and labeling.

A Series consists of two main components as in Fig 6.2.1.

1. **Data** – the array containing the actual values.
2. **Index** – an array of labels corresponding to the data values.



Index	Data
0	Mark
1	Justin
2	John
3	Vicky

Fig 6.2.1 Components of Series

In essence, a Series is a labeled, one-dimensional array that can store any data type.

Key characteristics include:

- The **data** within a Series is **mutable**, meaning its values can be modified.

- The **size** of the Series is **immutable,** so the number of elements cannot be changed after creation.
- A Series functions as a data structure composed of two arrays: one for the data and the other for its labels or index.
- The labels used to identify rows in a Series are referred to as the **Index.**

6.2.2.1 Creating a Pandas Series

A pandas Series can be created using the following constructor

class pandas.Series(data, index, dtype, name, copy)

Parameters of Constructor are

1. **data**-The input can be in different formats such as an ndarray, a list, or a single constant value.
2. **index**-The index must contain unique, hashable values and must match the length of the data. If not provided, it defaults to `np.arange(n)`.
3. **dtype**-Refers to the data type. If not specified, pandas will automatically detect it.
4. **copy**-Indicates whether to create a copy of the input data. The default setting is `False`.

Create an Empty Series

An empty Series object can be created by calling the pandas.Series() constructor without passing any data.

```
import pandas as pd                    #import the pandas library and aliasing as pd

s = pd.Series()

print('Resultant Empty Series:\n',s)          # Display the result
```

Create a Series from Python Dictionary

You can create a Series by passing a dictionary to the `pd.Series()` constructor. If no index is provided, the keys of the dictionary will be sorted and used as the Series index. If an index is specified, only the values that match the given labels will be extracted from the dictionary.

```
import pandas as pd

import numpy as np

data = {'a' : 0., 'b' : 1., 'c' : 2.}

s = pd.Series(data)

print(s)
```

6.2.3 DataFrame

A DataFrame is a two-dimensional structure as in Fig 6.2.2 ideal for organizing data in rows and columns, much like a spreadsheet or SQL table. It is the most widely used object in pandas. After loading data into a DataFrame, a variety of operations can be performed to explore, analyze, and interpret the data effectively.

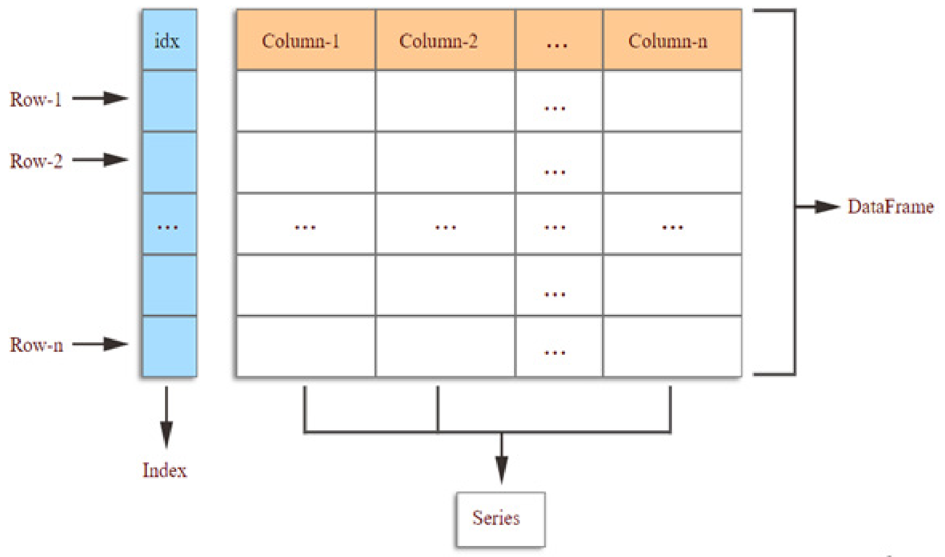


Fig 6.2.2 DataFrames

6.2.3.1 Properties of DataFrames

1. A DataFrame has two axes: the row axis (`axis=0`) and the column axis (`axis=1`).
2. It resembles a spreadsheet, where the row identifiers are called **indices** and the column identifiers are called **column names**.
3. It can store **heterogeneous data**, meaning different data types in different columns.
4. The **size of a DataFrame is mutable**, meaning rows and columns can be added or removed.
5. The **data within a DataFrame is also mutable**, so values can be changed after creation.

6.2.3.2 Creating a Pandas DataFrame

Create an Empty DataFrame

An empty DataFrame by calling the DataFrame constructor without passing any arguments.

```
import pandas as pd
df = pd.DataFrame()
```

```
print(df)
```

Create a DataFrame from Lists

A DataFrame can be constructed from a single list or from multiple lists organized within a list (i.e., a list of lists).

```
import pandas as pd

data = [1,2,3,4,5]

df = pd.DataFrame(data)

print(df)
```

Creating DataFrame from dict of ndarray/lists

A DataFrame can be constructed using a dictionary, where each key represents a column name and the corresponding value is a list or array of data.

- All lists or arrays used must be of equal length.
- If you specify an index, its length must also match the length of the data.
- If no index is given, Pandas will automatically assign a default integer index starting from 0.

```
import pandas as pd

data = {'Name':['Tom', 'nick', 'krish', 'jack'],

    'Age':[20, 21, 19, 18]}

df = pd.DataFrame(data)

print(df)
```

6.2.3.3 DataFrame Operations

1. Selection

Selecting specific rows or columns from a DataFrame using labels (`loc[]`) or index positions (`iloc[]`).

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

    'Age': [25, 30, 35],

    'City': ['NY', 'LA', 'Chicago']}

df = pd.DataFrame(data)

print(df['Name'])                    # Selecting a single column

print(df[['Name', 'City']])           # Selecting multiple columns
print(df.loc[1])  # Bob’s row                    # Selecting a row by index
```

Filtering

Extracting rows that meet specific conditions using boolean expressions.

```
import pandas as pd
data = {‘Name’: [‘Alice’, ‘Bob’, ‘Charlie’],
        ‘Age’: [25, 30, 35],
        ‘Salary’: [50000, 60000, 70000]}
df = pd.DataFrame(data)
filtered_df = df[df[‘Age’] > 28]              # Filtering rows where Age > 28
print(filtered_df)
```

Sorting

Arranging rows in ascending or descending order based on column values.

```
import pandas as pd
data = {‘Name’: [‘Alice’, ‘Bob’, ‘Charlie’],
        ‘Age’: [25, 30, 35]}
df = pd.DataFrame(data)
print(df.sort_values(by=’Age’))                         # Sorting by Age (ascending)
print(df.sort_values(by=’Age’, ascending=False))     # Sorting by Age (descending)
```

Merging

Combining two DataFrames using a common column or key.

```
import pandas as pd
df1 = pd.DataFrame({
    ‘ID’: [1, 2, 3],
    ‘Name’: [‘Alice’, ‘Bob’, ‘Charlie’]
})
df2 = pd.DataFrame({
    ‘ID’: [1, 2, 3],
    ‘Salary’: [50000, 60000, 70000]
})
```

```
merged_df = pd.merge(df1, df2, on='ID')       # Merging both DataFrames on ID

print(merged_df)
```

6.2.4 Handling Missing Data and Basic Data Cleaning

Missing data arises when information is unavailable for one or more elements, or even for an entire row or column. In Pandas, this is represented as `NaN` (Not a Number). In real-world datasets, incomplete data is common and can cause issues. Fortunately, Pandas offers various methods to manage and handle missing data efficiently.

The purpose of data cleaning is to correct or remove problematic data to ensure the dataset is accurate, consistent, and usable. Common issues that require cleaning include:

- Empty cells (missing values)
- Incorrect data formats (e.g., text instead of numbers)
- Duplicate records
- Logically incorrect data (e.g., negative age or future birthdate)

Cleaning this "bad data" is crucial because uncleaned data can result in flawed analyses, incorrect models, and misleading outputs.

1. Checking for Missing Values with `isnull()` and `notnull()`

To identify missing values (represented as `NaN`), Pandas provides two helpful functions:

- **`isnull()`**: Returns `True` for missing (`NaN`) entries and `False` for all others.
- **`notnull()`**: Returns `True` for non-missing values and `False` where data is missing.

```
import pandas as pd

import numpy as np

dict = {'First Score':[100, 90, np.nan, 95],

        'Second Score': [30, 45, 56, np.nan],

        'Third Score':[np.nan, 40, 80, 98]}

df = pd.DataFrame(dict)

df.isnull()
```

2. Filling Missing Values using fillna(), replace() and interpolate()

To handle missing (null) values in a dataset, Pandas offers functions like `fillna()`, `replace()`, and `interpolate()`. These functions allow you to substitute `NaN` values with appropriate alternatives. While `fillna()` and `replace()` typically use specific values to fill the gaps, the `interpolate()` function uses different interpolation methods to estimate and fill missing data, rather than assigning a fixed value directly. These tools are essential for managing null values within a DataFrame.

```
import pandas as pd

import numpy as np

dict = {'First Score':[100, 90, np.nan, 95],

        'Second Score': [30, 45, 56, np.nan],

        'Third Score':[np.nan, 40, 80, 98]}

df = pd.DataFrame(dict)

df.fillna(0)
```

3. Dropping Missing Values using dropna()

To eliminate rows or columns containing missing data, we can use the `dropna()` method. This method is versatile and can be configured to remove either rows or columns based on specific criteria.

```
import pandas as pd

import numpy as np

dict = {'First Score':[100, 90, np.nan, 95],

        'Second Score': [30, np.nan, 45, 56],

        'Third Score':[52, 40, 80, 98],

        'Fourth Score':[np.nan, np.nan, np.nan, 65]}

df = pd.DataFrame(dict)

df.dropna()
```

6.2.5 Advantages of Pandas

1. Easy Data Handling

Pandas provides a simple and intuitive way to load, manipulate, and analyze data using concise and readable syntax.

2. Efficient Data Structures

Pandas offers powerful data structures such as Series (for one-dimensional data) and DataFrame (for two-dimensional tabular data), which are flexible and well-suited for handling structured data.

3. Handles Missing Data

Pandas has built-in methods to detect, remove, and fill missing values, allowing users to manage incomplete data effectively.

4. Powerful Data Selection and Filtering

Pandas makes it easy to select specific rows and columns, apply conditions, and filter data using functions like `.loc[]`, `.iloc[]`, and boolean indexing.

5. Supports Data Alignment and Reshaping

With Pandas, data can be automatically aligned during operations, and reshaping functions allow flexible reorganization of the data layout.

6. Data Aggregation and Grouping

Pandas allows users to efficiently group and summarize data for analysis.

7. High Performance

Built on top of NumPy, Pandas is optimized for performance, especially for large datasets and numerical computations.

8. Rich I/O Support

Pandas supports reading from and writing to various file formats including CSV, Excel, JSON, SQL databases, and more, making it versatile for data import and export.

9. Integration with Other Libraries

Pandas integrates seamlessly with other popular Python libraries such as NumPy, Matplotlib, Scikit-learn, and TensorFlow, making it a core tool in the data science ecosystem.

10. Open Source and Actively Maintained

As an open-source library with a large and active community, Pandas is continuously improved and supported with regular updates and extensive documentation.

Recap

- Pandas is a powerful Python library for data analysis and manipulation.
- It provides two core data structures: **Series** (1D) and **DataFrame** (2D).
- A **Series** is a one-dimensional labeled array; data is mutable, size is fixed.
- A **DataFrame** is a two-dimensional table with rows and columns, like a spreadsheet.
- DataFrames can be created from lists, dictionaries, or arrays.
- Data can be selected using labels (`loc[]`) or positions (`iloc[]`).
- Filtering and sorting are easily done using conditions and `sort_values()`.
- DataFrames can be merged using `merge()` on common keys.

- Missing data is represented as `NaN` and can be detected using `isnull()` and `notnull()`.
- Missing values can be filled with `fillna()`, replaced with `replace()`, or estimated using `interpolate()`.
- Rows or columns with missing values can be removed using `dropna()`.
- Pandas supports essential data cleaning tasks like removing duplicates and fixing formats.
- It is fast, flexible, and integrates well with other Python libraries.

Objective Type Questions

1. Which Python library is commonly used for data analysis and manipulation?
2. What data structure in Pandas represents one-dimensional labeled data?
3. What Pandas structure resembles a spreadsheet or SQL table?
4. Which function is used to detect missing values in a DataFrame?
5. What value is used in Pandas to represent missing data?
6. Which method replaces missing values using interpolation techniques?
7. Which method is used to merge two DataFrames?
8. Which method removes rows or columns with missing data?
9. What constructor is used to create an empty Series?
10. Which method is used to sort data in a DataFrame?

Answers to Objective Type Questions

1. Pandas
2. Series
3. DataFrame
4. isnull
5. NaN

6. interpolate
7. merge
8. dropna
9. Series
10. sort_values

Assignments

1. Explain the difference between Series and DataFrame in Pandas with examples.
2. Write a Python program to create a DataFrame using a dictionary of lists and display its structure.
3. Demonstrate how to detect and handle missing data using `isnull()`, `fillna()`, and `dropna()` methods.
4. Write a program to create a Series from a dictionary and access elements using labels.
5. How can you filter rows in a DataFrame based on a condition? Provide code to demonstrate filtering.

Reference

1. VanderPlas, J. (2016). *Python data science handbook*. O'Reilly Media.
2. Chen, D. Y. (2017). *Pandas for everyone: Python data analysis*. Addison-Wesley.
3. Molin, S. (2019). *Hands-on data analysis with Pandas*. Packt Publishing.
4. Kazil, J., & Jarmul, K. (2016). *Data wrangling with Pandas*. O'Reilly Media.
5. McKinney, W. (2017). *Python for data analysis* (2nd ed.). O'Reilly Media.

Suggested Reading

1. https://numpy.org/doc/
2. https://www.w3schools.com/python/pandas/default.asp
3. https://www.geeksforgeeks.org/python-pandas-tutorial/

Unit 3

Data Visualization with Matplotlib

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- familiarize with the basic components of Matplotlib including Figure, Axes, and plot types.
- explore methods to create line charts, bar charts, and pie charts using Python code.
- describe how to customize plots by adding titles, labels, legends, colors, and markers.

Prerequisites

In your earlier programming journey, you might have worked with numerical and textual data - storing them in variables, processing them through logic, and even organizing them using structures like lists or arrays. You've likely learned how to perform operations using tools like NumPy or simple Python loops to analyze or transform this data. But after reaching a certain point, especially when dealing with large sets of values or time-based trends, a natural question arises: *How do we make sense of all this information at a glance?* That's where data visualization becomes essential.

You may also have seen tables or raw output printed on the screen and realized that while it's informative, it's not always intuitive. Visual representation helps us quickly interpret relationships, detect patterns, and make comparisons. This is particularly important in data science, business analysis, or any field involving decision-making. Just imagine trying to spot a trend in monthly sales or understand user growth by only reading a table - it's not impossible, but it's not easy either.

To bridge this gap, we now introduce Matplotlib - a powerful library in Python that allows us to turn numbers into visuals. By learning how to create line graphs, bar charts, and pie charts, you'll be able to give life to your data, making your analyses not just functional but also visually compelling. Let's now step into the world of plotting, where numbers meet narratives.

Key words

Matplotlib, Data Visualization, Line Chart, Bar Chart, Pie Chart, Axes

Discussion

6.3.1 Introduction to Data Visualization

Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data.

Matplotlib is one of the most popular Python libraries used for data visualization. It allows users to create a wide range of static, animated, and interactive plots with simple commands.

6.3.2 Introduction to Matplotlib

Matplotlib is a comprehensive library in Python that enables users to create static, animated, and interactive visualizations with ease. It was created by John Hunter in 2003 with the aim of providing a plotting tool for Python users that closely resembled MATLAB in style and functionality. This design choice made it familiar and accessible to users who were already experienced with MATLAB. Matplotlib integrates seamlessly with other scientific computing libraries like NumPy and Pandas, making it a powerful tool for data analysis and visualization in the Python ecosystem.

Before using Matplotlib, you need to install it in your system. This is done using the Python package installer **pip**.

To install:

1. Open your command prompt (Windows) or terminal (Mac/Linux).
2. Type the below command and press Enter.

```
pip install matplotlib
```

This tells Python to download and install the latest version of the Matplotlib library from the Python Package Index (PyPI). Make sure you have internet access while running this command.

To import:

Once the installation is successful, you need to import the library into your Python script to use it.

```
import matplotlib.pyplot as plt
```

- ♦ matplotlib.pyplot is a module within Matplotlib that provides a MATLAB-like interface for plotting.
- ♦ plt is a commonly used alias that makes it easier to refer to the module throughout your code.
- ♦ After importing, you can use plt to call functions like plt.plot(), plt.title(), plt.show(), etc., to create and display plots.

6.3.3 The pyplot Module

pyplot is a collection of functions that make Matplotlib work like MATLAB. Each function makes some change to a figure.

Example:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]

y = [10, 20, 25, 30]

plt.plot(x, y)

plt.show()
```

This code plots a simple line chart.

Explanation:

- x = [1, 2, 3, 4]: These are the x-axis values.
- y = [10, 20, 25, 30]: These are the y-axis values.
- plt.plot(x, y): This draws a line connecting the points (1,10), (2,20), (3,25), and (4,30).
- plt.show(): This displays the plot in a window.

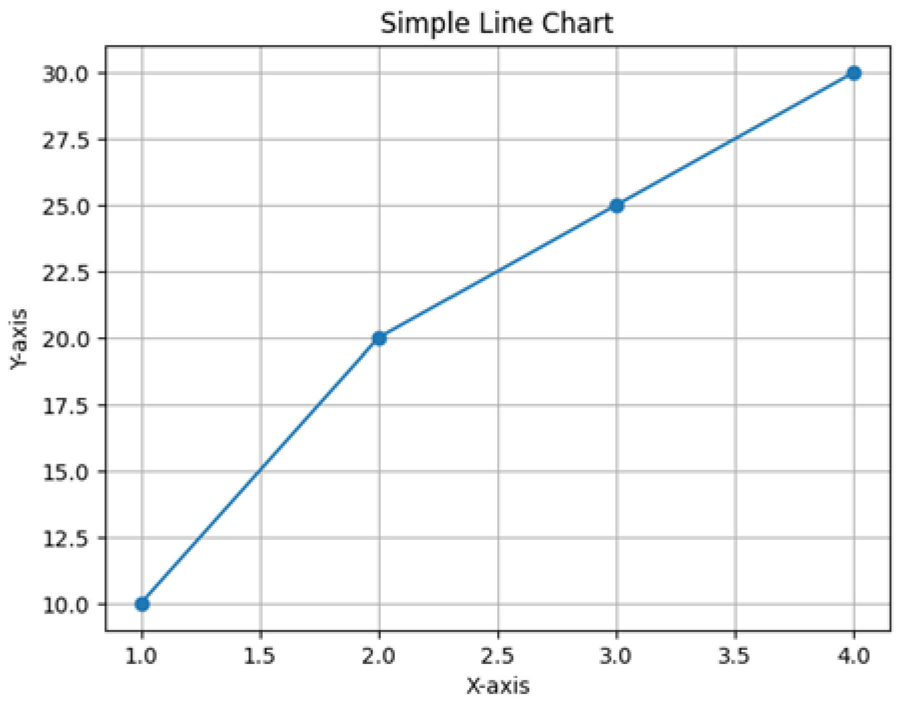


Fig. 6.3.1 Graph of the points (1,10), (2,20), (3,25), and (4,30), displayed as a line chart with markers on each point.

6.3.4 The Axes Class Methods

In Matplotlib, the **Axes** is the part of the figure where the actual data is drawn — such as lines, bars, or pies. Think of a figure as a piece of paper, and an axes as the rectangular area on that paper where the chart appears.

Using the Axes class, we can create detailed and customized plots by controlling how data appears inside that plotting area.

You usually create Axes using functions like plt.subplots() or fig.add_subplot().

Common Axes Methods and Their Usage

Let's explore some of the frequently used Axes methods:

- **ax.plot() — Draws Line Charts**

This method is used to draw line plots.

Example:

ax.plot([1, 2, 3], [4, 5, 6])

This draws a line connecting the points (1,4), (2,5), and (3,6).

- **ax.bar() — Draws Bar Charts**

This method creates vertical bars for categorical data.

Example:

ax.bar(['A', 'B', 'C'], [10, 20, 15])

This draws bars labeled A, B, and C with heights 10, 20, and 15 respectively.

- **ax.pie() — Draws Pie Charts**

Used to create circular pie charts to show percentage or proportional data.

Example:

ax.pie([30, 40, 30], labels=['A', 'B', 'C'])

This creates a pie chart divided into three labeled parts.

- **ax.set_title() — Adds a Title**

Sets the title at the top of the plot.

Example:

ax.set_title("Sales Over Time")

- **ax.set_xlabel() — Labels the X-axis**

Adds a label below the x-axis.

Example:

ax.set_xlabel("Months")

- **ax.set_ylabel() — Labels the Y-axis**

Adds a label beside the y-axis.

Example:

ax.set_ylabel("Revenue in USD")

6.3.5 The Figure Class Methods

In Matplotlib, a Figure is like a blank canvas or an entire page where one or more plots (called Axes) can be drawn. Think of it as the outer frame that holds everything - including axes, labels, legends, titles, and possibly multiple plots arranged in a grid.

Each figure can have one or more Axes (plots), and each Axes represents a single chart area where data is actually plotted.

Commonly Used Figure Methods

- **figure()**

This method creates a new empty figure object.

Usage:

fig = plt.figure()

Now, fig represents the entire canvas.

- **add_subplot()**

This adds a subplot (Axes) to the figure. You can arrange subplots in a grid layout using three numbers: rows, columns, and position.

Usage:

ax1 = fig.add_subplot(1, 1, 1)

This means: "Add one subplot in a 1-row, 1-column grid — and this is the first one."

- **savefig()**

This saves the current figure as an image file (like PNG, JPG, PDF, etc.).

Usage:

plt.savefig("plot.png")

This saves the entire figure - not just one plot - as "plot.png" in the current working directory.

6.3.6 Creating Line Charts

Line charts are one of the most commonly used visualizations in data analysis. They are especially useful for:

- Showing trends over time
- Tracking changes in values
- Comparing one or more series of data

Each point in a line chart is connected by a line, making it easy to see upward or downward trends.

Example:

```
        import matplotlib.pyplot as plt
# Data
months = ['January', 'February', 'March', 'April']
sales = [1500, 1800, 1700, 2100]
# Basic line chart
plt.plot(months, sales)
# Show the plot
plt.show()
```

Output:

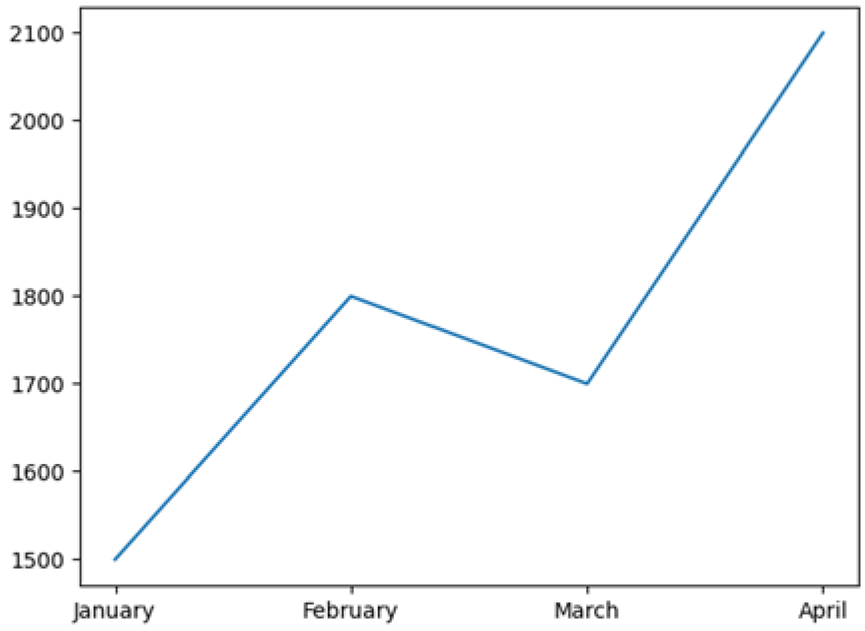


Fig 6.3.7 Creating Bar Charts

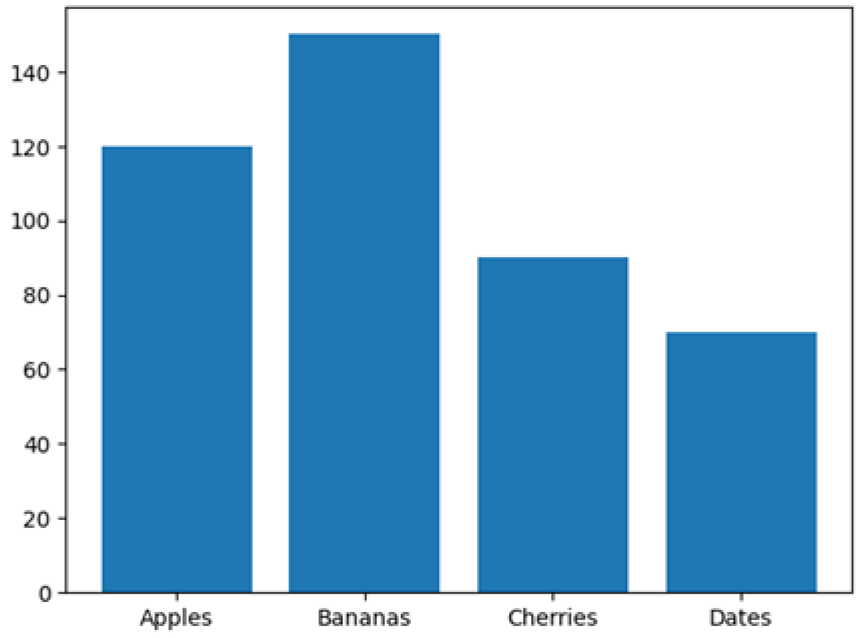
A bar chart is used to display categorical data with rectangular bars. Each bar's height (or length) represents the value of that category. It's ideal for comparing values across different groups.

- **Use Cases:**
  - Comparing sales across different products
  - Displaying counts of items in categories
  - Visualizing survey results

Example:

```
        import matplotlib.pyplot as plt
# Data
x = ['Apples', 'Bananas', 'Cherries', 'Dates']
y = [120, 150, 90, 70]
# Create bar chart
plt.bar(x, y)
# Show the plot
plt.show()
```

Output:



What You See:

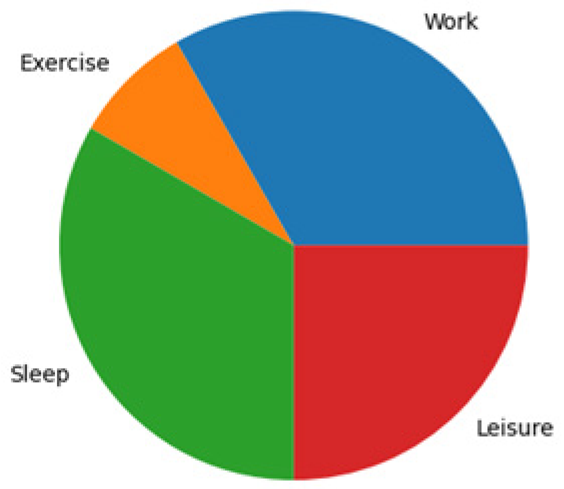
- A vertical bar chart with 4 bars.
- Each bar represents a type of fruit.
- The height of each bar corresponds to the sales quantity:
    - Apples: 120 units
    - Bananas: 150 units (tallest bar)
    - Cherries: 90 units
    - Dates: 70 units (shortest bar)
- The x-axis shows the fruit names, and the y-axis shows numerical values (automatically scaled based on the data)

6.3.8 Creating Pie Charts

Example:

```
import matplotlib.pyplot as plt

# Data

labels = ['Work', 'Exercise', 'Sleep', 'Leisure']

sizes = [8, 2, 8, 6]

# Create pie chart

plt.pie(sizes, labels=labels)

# Show the plot

plt.show()
```

Output:



What You See:

- A circular pie chart divided into 4 slices.
- Each slice is labeled with an activity: Work, Exercise, Sleep, and Leisure.
- The size of each slice reflects the number of hours spent per day:
  - Work and Sleep have equal, larger slices (8 hours each).
  - Leisure is slightly smaller (6 hours).
  - Exercise is the smallest (2 hours).
- The chart starts from the right and proceeds counterclockwise (default behavior)

6.3.9 Plot Customization

Customization helps make your plots **clearer**, **more informative**, and **visually appealing**. You can control how the chart looks by modifying titles, labels, colors, and styles.

6.3.9.1 Adding Titles and Axis Labels

Titles and labels make it easier to understand what the chart is showing.

```
plt.title("Custom Title")      # Adds a title at the top of the plot
plt.xlabel("X Label")          # Adds a label below the x-axis
plt.ylabel("Y Label")          # Adds a label beside the y-axis
```

Example:

```
plt.plot([1, 2, 3], [4, 5, 6])
plt.title("Simple Line Plot")
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
```

6.3.9.2 Adding a Legend

When you plot multiple lines, a legend helps identify which line represents which dataset.

```
plt.plot(x, y, label="My Line")  # Label this line
plt.legend()                     # Show the legend box
```

Example:

```
x = [1, 2, 3]
y = [4, 5, 6]
```

plt.plot(x, y, label=”Experiment A”)

plt.legend()

6.3.9.3 Changing Colors

You can change the color of the line or markers using the color parameter.

plt.plot(x, y, color=’green’)   # Changes the line color to green

Common Colors:

- ‘red’, ‘blue’, ‘green’, ‘orange’, ‘purple’
- You can also use short codes: ‘r’ for red, ‘b’ for blue, etc.

6.3.9.4 Changing Line Styles and Markers

You can make lines dashed or dotted, and use markers (symbols) for each data point.

plt.plot(x, y, linestyle=’--’, marker=’o’, color=’blue’)

Parameters:

- linestyle=’--’ → Dashed line
- marker=’o’ → Circles at each data point
- color=’blue’ → Blue color for the line

Other Marker Options:

‘o’: Circle

‘s’: Square

‘^’: Triangle

‘*’: Star

Other Line Styles:

- ‘-’: Solid line (default)
- ‘--’: Dashed
- ‘:’: Dotted
- ‘-.’: Dash-dot

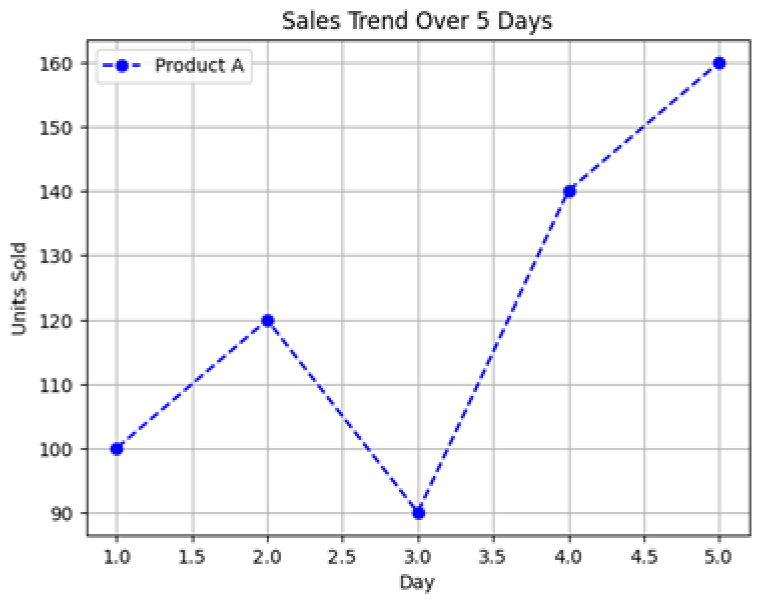
Example using all Customizations
import matplotlib.pyplot as plt

# Sample data

x = [1, 2, 3, 4, 5]

```
y = [100, 120, 90, 140, 160]
# Create a customized line chart
plt.plot(x, y, label=”Product A”, color=’blue’, linestyle=’--’, marker=’o’)
# Add title and axis labels
plt.title(“Sales Trend Over 5 Days”)
plt.xlabel(“Day”)
plt.ylabel(“Units Sold”)
# Show legend
plt.legend()
# Add grid
plt.grid(True)
# Show the plot
plt.show()
```

Output:



What you will see:

- A blue dashed line with circular markers
- A legend showing the label “Product A”
- A title at the top: “Sales Trend Over 5 Days”
- Labels on both the x-axis (“Day”) and y-axis (“Units Sold”)

- A **grid** to make reading values easier

Saving Figures

You can save any figure using:

plt.savefig("my_plot.png")

You can also specify file formats like .jpg, .svg, .pdf, etc.

Recap

- Matplotlib is used for data visualization in Python
- Created by John Hunter in 2003
- Works well with NumPy and Pandas
- Designed to mimic MATLAB-style plotting

Basic Plot Types

- plt.plot() → Line chart
- plt.bar() → Bar chart
- plt.pie() → Pie chart

Line Chart

- Shows trends over time or sequences
- x and y values required
- Basic: plt.plot(x, y)
- Add marker, linestyle, color for customization

Bar Chart

- Used for comparing categories
- plt.bar(x, y) → vertical bars
- Categories on x-axis, values on y-axis

Pie Chart

- Used for showing parts of a whole
- Needs values and labels

- plt.pie(values, labels=labels)

Figure and Axes

- plt.figure() → creates a figure (canvas)
- fig.add_subplot() → adds subplot (Axes)
- plt.savefig() → saves the figure to a file

Axes Class Methods

- ax.plot() → Line plot on axes
- ax.set_title() → Title
- ax.set_xlabel(), ax.set_ylabel() → Axis labels

Plot Customization

- plt.title(), plt.xlabel(), plt.ylabel()
- label= → used in plot() for legends
- plt.legend() → displays legend box
- color= → changes line color
- linestyle= → solid, dashed, dotted, etc.
- marker= → marks data points (circle, square, triangle, etc.)
- plt.grid(True) → adds background grid

Objective Type Questions

1. Which Python library is commonly used for plotting graphs?
2. What function is used to draw a line chart?
3. Which method saves a figure to a file?
4. What chart type is best to show a whole divided into parts?
5. Which method is used to add a title to the plot?
6. What argument in plot() changes the color of the line?
7. What type of chart uses vertical bars to represent data?
8. What keyword in plot() adds a label for the legend?

9. Which function is used to display the plot window?
10. What method is used to add a subplot to a figure?
11. Which argument is used to mark data points in a line chart?
12. Which keyword enables background gridlines?
13. What is the method to create a new figure in Matplotlib?
14. What attribute of a plot is set using xlabel()?
15. What does plt.pie() require in addition to values?

Answers to Objective Type Questions

1. Matplotlib
2. plot
3. savefig
4. Pie
5. title
6. color
7. Bar
8. label
9. show
10. add_subplot
11. marker
12. grid
13. figure
14. X-axis label
15. labels

Assignments

1. Plot a line chart showing temperature change over a week.
2. Create a bar chart comparing the number of students in four different departments.
3. Make a pie chart representing budget allocation for an event.
4. Customize a chart with labels, title, and legend.
5. Create a subplot with one line chart and one bar chart.
6. Explain the structure of Matplotlib's figure and axes hierarchy with a program that uses figure() and add_subplot() methods.
7. Create a customized line chart using Matplotlib. Include title, axis labels, grid, legend, color, line style, and markers. Explain how each customization affects the chart.

Reference

1. Hunter, J. D. (2007). *Matplotlib: A 2D graphics environment*. *Computing in Science & Engineering*, 9(3), 90–95. https://doi.org/10.1109/MCSE.2007.55
2. VanderPlas, J. (2016). *Python Data Science Handbook*. O'Reilly Media.
3. McKinney, W. (2017). *Python for Data Analysis* (2nd ed.). O'Reilly Media.
4. Matplotlib Developers. (2024). *Matplotlib Documentation*. Retrieved from https://matplotlib.org/stable/

Suggested Reading

1. Swaroop, C. H. (2019). *Python Programming: A Modern Approach*. Oxford University Press.
2. Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.
3. Ascher, D., & Lutz, M. (2003). *Learning Python* (2nd ed.). O'Reilly Media.
4. Yadav, T. (2021). *Data Science with Python*. BPB Publications.

Unit 4

Data Aggregation and Advanced Visualization

Learning Outcomes

After the successful completion of the course, the learner will be able to:

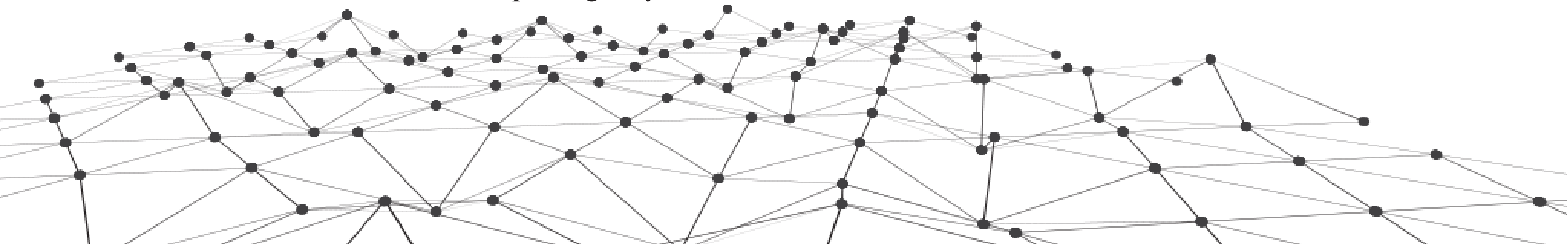
- recall basic data aggregation functions in Pandas.
- identify differences between aggregation and transformation.
- construct simple scatter plots and histograms.
- label visualization components accurately.

Prerequisites

In today's data-driven world, the ability to make sense of vast datasets is a superpower. Raw data, in its unaggregated form, can often be overwhelming—a sea of individual transactions, measurements, or events with little immediate meaning. This is where the crucial skills of data aggregation and analysis come in. By learning to group, summarize, and transform raw data, you gain the power to uncover hidden patterns, identify key trends, and extract the insights that drive business decisions. These assignments are designed to be your first step in mastering this process, teaching you how to turn a large, complex spreadsheet into a concise, powerful report that tells a clear story.

As you work through these problems, you will become proficient with the indispensable tools of the trade: Pandas' groupby(), pivot_table(), and crosstab(). These are not just functions; they are the fundamental building blocks for any data analyst or data scientist. They provide you with the means to effortlessly slice and dice your data, whether you need to find total sales per product, average price per customer, or the frequency of purchases across different categories. This hands-on experience will solidify your understanding and prepare you to tackle real-world data challenges with confidence.

Finally, you will move beyond simple numbers and learn to visualize your findings. After you have aggregated the data, visualization is the most effective way to communicate your discoveries. The scatter plot assignments will teach you how to visually represent relationships between variables, allowing you to spot correlations and outliers that would be invisible in a raw table of numbers. Completing these assignments will equip you with a foundational toolkit for both crunching numbers and presenting your results in a clear, compelling way.



Key words

Pandas, Data Aggregation, Pivot Table, Cross-tabulation, Scatter Plot

Discussion

6.4.1 Introduction to Data Aggregation

Data aggregation is the fundamental process of gathering raw data and summarizing it into a more concise and meaningful form. In the realm of data science, this process is indispensable for transforming large, complex datasets into actionable insights. By reducing the volume of data while preserving its essential characteristics, aggregation facilitates easier analysis, identification of trends, and decision-making. Imagine trying to analyze individual sales transactions for an entire year across thousands of stores; it would be an overwhelming task. Data aggregation allows us to summarize these transactions by region, product category, or even hourly sales, making the information digestible and revealing overall patterns.

Example:

Consider a dataset containing individual customer transactions at an online retail store. Each row represents a single purchase and includes information like CustomerID, ProductID, PurchaseDate, Quantity, and Price.

CustomerID	ProductID	PurchaseDate	Quantity	Price
1001	A101	2024-07-01	2	25.00
1002	B205	2024-07-01	1	50.00
1001	C303	2024-07-02	1	10.00
1003	A101	2024-07-02	3	25.00
1002	A101	2024-07-03	1	25.00

Without aggregation, analyzing total sales per day or identifying the most popular products would require manual summation, which is prone to error and highly inefficient for large datasets. Data aggregation techniques allow us to quickly calculate, for instance, the total revenue generated each day or the total quantity sold for each product.

6.4.2 Understanding Data Grouping in Pandas

Pandas, a powerful open-source data analysis and manipulation library for Python, provides highly optimized tools for data grouping. The groupby() method is central to this process. It allows you to split a DataFrame into groups based on one or more criteria, typically the unique values within a column or set of columns. Once grouped, you can apply various operations to each group independently.

The groupby() operation involves three main steps:

1. Splitting: The data is divided into groups based on the specified keys (e.g., column values).
2. Applying: A function (an aggregation, transformation, or filtration) is applied to each group.
3. Combining: The results of the applied function are combined into a new data structure.

Example:

Let's continue with our retail sales data. We want to understand the total sales for each ProductID.

```
import pandas as pd

data = {

   'CustomerID': [1001, 1002, 1001, 1003, 1002, 1001, 1004],

   'ProductID': ['A101', 'B205', 'C303', 'A101', 'A101', 'B205', 'C303'],

    'PurchaseDate': ['2024-07-01', '2024-07-01', '2024-07-02', '2024-07-02', '2024-
07-03', '2024-07-03', '2024-07-04'],

   'Quantity': [2, 1, 1, 3, 1, 2, 1],

   'Price': [25.00, 50.00, 10.00, 25.00, 25.00, 50.00, 10.00]

}

df = pd.DataFrame(data)

# Group by 'ProductID'

grouped_by_product = df.groupby('ProductID')

print("Type of grouped_by_product:", type(grouped_by_product))

print("\nGroups in grouped_by_product:")

for name, group in grouped_by_product:

   print(f"\nProduct: {name}")

   print(group)
```

Output (conceptual, as printing the GroupBy object directly shows memory address):

```
Type of grouped_by_product: <class 'pandas.core.groupby.generic.DataFrameGroupBy'>

Groups in grouped_by_product:

Product: A101
   CustomerID ProductID PurchaseDate  Quantity  Price
0        1001      A101   2024-07-01         2   25.0
3        1003      A101   2024-07-02         3   25.0
4        1002      A101   2024-07-03         1   25.0

Product: B205
   CustomerID ProductID PurchaseDate  Quantity  Price
1        1002      B205   2024-07-01         1   50.0
5        1001      B205   2024-07-03         2   50.0

Product: C303
   CustomerID ProductID PurchaseDate  Quantity  Price
2        1001      C303   2024-07-02         1   10.0
6        1004      C303   2024-07-04         1   10.0
```

This output shows that grouped_by_product is a DataFrameGroupBy object, which conceptually holds separate DataFrames for each unique ProductID.

6.4.3 Aggregation Functions in Pandas

Once data is grouped using groupby(), various aggregation functions can be applied to each group to summarize the data. Pandas provides a rich set of built-in functions that are commonly used for this purpose.

Common aggregation functions include:

1. sum(): Calculates the sum of values in each group.
2. mean(): Computes the average of values in each group.
3. count(): Counts the number of non-null values in each group.
4. min(): Finds the minimum value in each group.
5. max(): Finds the maximum value in each group.
6. median(): Calculates the median of values in each group.
7. std(): Computes the standard deviation of values in each group.
8. var(): Computes the variance of values in each group.

These functions are applied directly to the GroupBy object, and Pandas intelligently applies them to the appropriate numeric columns within each group.

Example:

Continuing with our sales data, let's calculate the total Quantity sold and the mean Price for each ProductID.

# Calculate total quantity sold for each product

```
total_quantity_per_product = grouped_by_product[‘Quantity’].sum()

print(“\nTotal Quantity Sold per Product:”)

print(total_quantity_per_product)

# Calculate average price per product (note: average price of items sold, not product price)

average_price_per_product = grouped_by_product[‘Price’].mean()

print(“\nAverage Price of Transactions per Product:”)

print(average_price_per_product)

# We can also apply directly to the grouped object, which aggregates all applicable columns

summary_stats_per_product = grouped_by_product.sum() # Sums all numeric columns

print(“\nSummary Statistics (Sum) per Product:”)

print(summary_stats_per_product)

# Example of counting transactions per product

transactions_per_product = grouped_by_product[‘CustomerID’].count()

print(“\nNumber of Transactions per Product:”)

print(transactions_per_product)
```

Output:

```
Total Quantity Sold per Product:
ProductID
A101    6
B205    3
C303    2
Name: Quantity, dtype: int64

Average Price of Transactions per Product:
ProductID
A101    25.0
B205    50.0
C303    10.0
Name: Price, dtype: float64

Summary Statistics (Sum) per Product:
           CustomerID                   PurchaseDate  Quantity  Price
ProductID
A101             3006  2024-07-012024-07-022024-07-03         6   75.0
B205             2003            2024-07-012024-07-03         3  100.0
C303             2005            2024-07-022024-07-04         2   20.0

Number of Transactions per Product:
ProductID
A101    3
B205    2
C303    2
Name: CustomerID, dtype: int64
```

These results clearly show aggregated insights, such as ProductID 'A101' having the highest total quantity sold (6 units) and ProductID 'B205' having an average transaction price of $50.00.

6.4.4 GroupBy Object and its Operations

The result of a groupby() call is not a DataFrame itself, but a special GroupBy object. This object is an intermediate structure that efficiently stores information about how the original DataFrame has been split into groups. It's a "lazy" object, meaning the actual computations (like aggregation) are not performed until an aggregation function is called on it.

The GroupBy object supports several powerful operations:

- Iteration: You can iterate over the GroupBy object to access each group as a tuple of (name, group_dataframe), where name is the group key and group_dataframe is a DataFrame containing the data for that group.
- Column Access: You can select one or more columns from the GroupBy object before applying an aggregation. This allows you to apply functions to specific columns only.
- Chaining: GroupBy operations can be chained with aggregation functions, making the code more concise and readable.

Example:

Let's demonstrate iterating through the GroupBy object and accessing specific columns.

```
# Accessing a specific group (e.g., product 'A101')

product_a101_group = grouped_by_product.get_group('A101')

print("\nData for Product A101:")

print(product_a101_group)

# Chaining operations: calculate the sum of 'Quantity' for each product directly

chained_sum = df.groupby('ProductID')['Quantity'].sum()

print("\nChained Sum of Quantity per Product:")

print(chained_sum)

# Chaining operations with multiple columns

chained_sum_price_quantity = df.groupby('ProductID')[['Quantity', 'Price']].sum()

print("\nChained Sum of Quantity and Price per Product:")

print(chained_sum_price_quantity)
```

Output:

```
Data for Product A101:
   CustomerID ProductID PurchaseDate  Quantity  Price
0        1001      A101   2024-07-01         2   25.0
3        1003      A101   2024-07-02         3   25.0
4        1002      A101   2024-07-03         1   25.0

Chained Sum of Quantity per Product:
ProductID
A101    6
B205    3
C303    2
Name: Quantity, dtype: int64

Chained Sum of Quantity and Price per Product:
           Quantity  Price
ProductID
A101              6   75.0
B205              3  100.0
C303              2   20.0
```

Understanding the GroupBy object is crucial because it clarifies how Pandas manages and processes grouped data efficiently, enabling flexible and powerful data manipulation.

6.4.5 Applying Multiple Aggregations

Often, you'll need to compute several different aggregate statistics for the same grouped data. Pandas' .agg() method is specifically designed for this purpose, allowing you to apply multiple aggregation functions simultaneously to one or more columns. This is far more efficient and concise than applying each aggregation function separately.

The .agg() method can take:

- A string or list of strings representing the names of the aggregation functions (e.g., 'sum', 'mean').
- A dictionary mapping column names to a single function or a list of functions to apply to that column.

Example:

Let's find the total Quantity sold, the average Price, and the number of distinct CustomerIDs for each ProductID.

```
import pandas as pd

# Sample data

data = {

   'ProductID': [101, 101, 102, 102, 103, 103, 103],
    'Quantity': [2, 4, 1, 3, 5, 2, 1],
    'Price': [100, 150, 200, 250, 300, 280, 270],
    'CustomerID': [1, 2, 1, 3, 2, 3, 4]
}
df = pd.DataFrame(data)
# 1. Apply multiple aggregations using a list of strings
multi_agg_list = df.groupby('ProductID').agg(['sum', 'mean', 'count'])
print("\nMultiple Aggregations (sum, mean, count) per Product:")
print(multi_agg_list)
# 2. Apply different aggregations to different columns using a dictionary
multi_agg_dict = df.groupby('ProductID').agg({
    'Quantity': 'sum',
    'Price': 'mean',
    'CustomerID': pd.Series.nunique  # You can also use 'nunique' as a string
})
multi_agg_dict.columns = ['TotalQuantity', 'AveragePrice', 'NumCustomers']
print("\nCustom Multiple Aggregations per Product:")
print(multi_agg_dict)
# 3. Combining aggregation and custom names (only one column at a time)
multi_agg_custom_names = df.groupby('ProductID').agg(
    total_qty=('Quantity', 'sum'),
    average_qty=('Quantity', 'mean'),
    min_qty=('Quantity', 'min'),
    max_qty=('Quantity', 'max')
)
print("\nMultiple Aggregations with Custom Names for Quantity:")
print(multi_agg_custom_names)
```

Output:

```
Multiple Aggregations (sum, mean, count) per Product:
          Quantity                Price                  CustomerID       \
               sum      mean count   sum        mean count        sum mean
ProductID
101              6  3.000000     2   250  125.000000     2          3  1.5
102              4  2.000000     2   450  225.000000     2          4  2.0
103              8  2.666667     3   850  283.333333     3          9  3.0


          count
ProductID
101           2
102           2
103           3

Custom Multiple Aggregations per Product:
           TotalQuantity  AveragePrice  NumCustomers
ProductID
101                    6    125.000000             2
102                    4    225.000000             2
103                    8    283.333333             3

Multiple Aggregations with Custom Names for Quantity:
           total_qty  average_qty  min_qty  max_qty
ProductID
101                6     3.000000        2        4
102                4     2.000000        1        3
103                8     2.666667        1        5
```

The first output using a list of strings ([‘sum’, ‘mean’, ‘count’]) results in a MultiIndex column, which can be useful but sometimes complex to work with. The second output, using a dictionary, is generally preferred as it allows for clear, custom names for the resulting aggregated columns, making the output much more readable and easier to use in subsequent analyses. Notice how CustomerID mean is inf in the first example; this is because CustomerID is not a meaningful numerical column for mean calculation in this context.

6.4.6 Pivot Tables in Pandas

Pivot tables are powerful tools for summarizing and reorganizing data in a tabular form, providing a multi-dimensional view of your dataset. They allow you to transform rows into columns, and vice versa, while performing aggregations on the data. The pivot_ table() function in Pandas is highly versatile and widely used for data exploration and reporting.

Key parameters of pd.pivot_table():

- ♦ data: The DataFrame to be pivoted.
- ♦ values: The column(s) to aggregate.
- ♦ index: The column(s) to use as new row labels.
- ♦ columns: The column(s) to use as new column labels.
- ♦ aggfunc: The aggregation function(s) to apply (e.g., ‘sum’, ‘mean’, or a list of functions).
- ♦ fill_value: Value to replace missing values (NaNs) in the pivot table.

- margins: If True, adds row/column subtotals/grand totals.

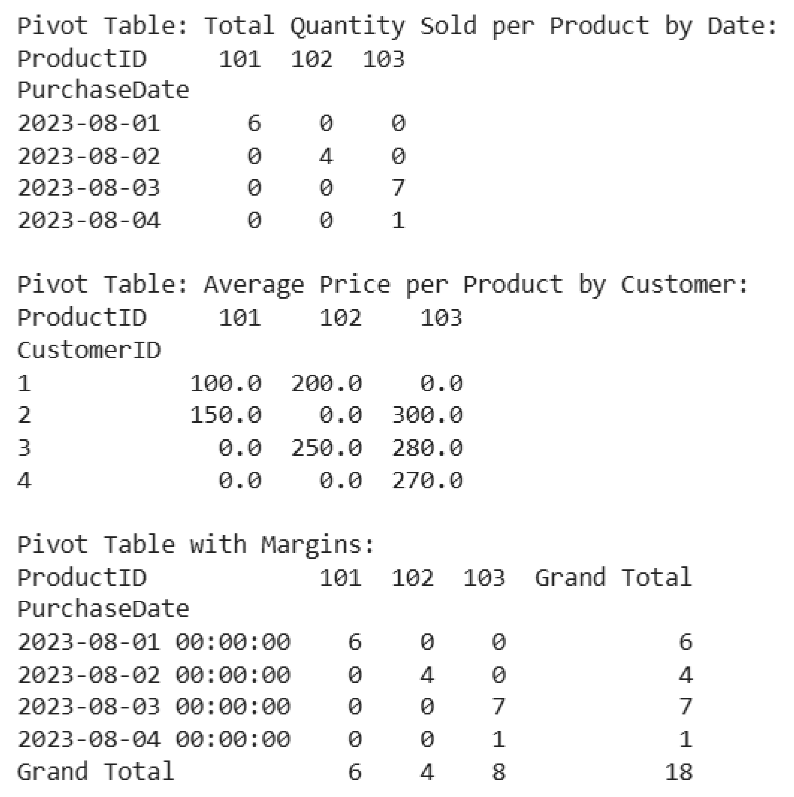
Example:

Let's create a pivot table to see the total Quantity sold per ProductID for each PurchaseDate.

```
import pandas as pd
# Sample data
data = {
   'ProductID': [101, 101, 102, 102, 103, 103, 103],
   'Quantity': [2, 4, 1, 3, 5, 2, 1],
   'Price': [100, 150, 200, 250, 300, 280, 270],
   'CustomerID': [1, 2, 1, 3, 2, 3, 4],
    'PurchaseDate': ['2023-08-01', '2023-08-01', '2023-08-02', '2023-08-02', '2023-
08-03', '2023-08-03', '2023-08-04']
}
df = pd.DataFrame(data)
# Convert 'PurchaseDate' to datetime
df['PurchaseDate'] = pd.to_datetime(df['PurchaseDate'])
# 1. Pivot table: total quantity sold by product and date
pivot_table_qty = pd.pivot_table(df,
                                values='Quantity',
                                index='PurchaseDate',
                                columns='ProductID',
                                aggfunc='sum',
                                fill_value=0)
print("\nPivot Table: Total Quantity Sold per Product by Date:")
print(pivot_table_qty)
# 2. Pivot table: average price per product by customer
pivot_table_avg_price = pd.pivot_table(df,
                                    values='Price',
                                    index='CustomerID',
                                    columns='ProductID',
                                    aggfunc='mean',
                                    fill_value=0)
```

```
print(“\nPivot Table: Average Price per Product by Customer:”)
print(pivot_table_avg_price)
# 3. Pivot table with margins (grand totals)
pivot_table_margins = pd.pivot_table(df,
                                     values=’Quantity’,
                                     index=’PurchaseDate’,
                                     columns=’ProductID’,
                                     aggfunc=’sum’,
                                     fill_value=0,
                                     margins=True,
                                     margins_name=’Grand Total’)
print(“\nPivot Table with Margins:”)
print(pivot_table_margins)
```

Output:



```
Pivot Table: Total Quantity Sold per Product by Date:
ProductID     101  102  103
PurchaseDate
2023-08-01      6    0    0
2023-08-02      0    4    0
2023-08-03      0    0    7
2023-08-04      0    0    1

Pivot Table: Average Price per Product by Customer:
ProductID     101    102    103
CustomerID
1           100.0  200.0    0.0
2           150.0    0.0  300.0
3             0.0  250.0  280.0
4             0.0    0.0  270.0

Pivot Table with Margins:
ProductID            101  102  103  Grand Total
PurchaseDate
2023-08-01 00:00:00    6    0    0            6
2023-08-02 00:00:00    0    4    0            4
2023-08-03 00:00:00    0    0    7            7
2023-08-04 00:00:00    0    0    1            1
Grand Total            6    4    8           18
```

The pivot tables present summarized data that is easy to read and interpret. For example, the first pivot table clearly shows that on 2024-07-02, ProductID ‘A101’ had 3 units sold, while ProductID ‘C303’ had 1 unit sold, and ProductID ‘B205’ had none. The margins provide quick grand totals for both rows and columns.

6.4.7 Cross-tabulation with pd.crosstab()

Cross-tabulation (often referred to as a "contingency table") is a specialized form of tabulation that displays the joint distribution of two or more categorical variables. It's used to count the frequency of occurrences for combinations of different categories. Pandas provides the pd.crosstab() function specifically for this purpose, making it an essential tool for categorical data analysis and understanding relationships between discrete variables.

Key parameters of pd.crosstab():

- index: The values to group by in the rows.
- columns: The values to group by in the columns.
- values: Optional, an array of values to aggregate according to the factors. If None, crosstab will count the frequencies.
- aggfunc: Optional, aggregate function if values is specified.
- margins: If True, adds row/column subtotals/grand totals.
- normalize: If True, normalizes by all (sum to 1), by row, or by column.

Example:

Let's examine the relationship between CustomerID and ProductID by counting how many times each customer purchased each product.

Python

```
# Cross-tabulation of CustomerID and ProductID
crosstab_customer_product = pd.crosstab(index=df['CustomerID'],
                                        columns=df['ProductID'])
print("\nCross-tabulation: Customer Purchases by Product:")
print(crosstab_customer_product)
# Cross-tabulation with margins
crosstab_margins = pd.crosstab(index=df['CustomerID'],
                               columns=df['ProductID'],
                               margins=True, # Add row/column totals
                               margins_name='Total')
print("\nCross-tabulation with Margins:")
print(crosstab_margins)
# Cross-tabulation showing the sum of Quantity for each combination
```

```
crosstab_sum_qty = pd.crosstab(index=df['CustomerID'],
                    columns=df['ProductID'],
                    values=df['Quantity'],
                    aggfunc='sum',
                    fill_value=0) # Fill missing combinations with 0
print("\nCross-tabulation: Total Quantity by Customer and Product:")
print(crosstab_sum_qty)
```

Output:

```
Cross-tabulation: Customer Purchases by Product:
ProductID   A101  B205  C303
CustomerID
1001           1     1     1
1002           1     1     0
1003           1     0     0
1004           0     0     1
Cross-tabulation with Margins:
ProductID  A101  B205  C303  Total
CustomerID
1001          1     1     1      3
1002          1     1     0      2
1003          1     0     0      1
1004          0     0     1      1
Total         3     2     2      8
Cross-tabulation: Total Quantity by Customer and Product:
ProductID   A101  B205  C303
CustomerID
1001           2     2     1
1002           1     1     0
1003           3     0     0
1004           0     0     1
```

The first cross-tabulation shows the count of transactions. For instance, CustomerID 1001 has purchased ProductID 'A101' once, ProductID 'B205' once, and ProductID 'C303' once. The second crosstab with margins indicates CustomerID 1001 made a total of 3 purchases across all products. The third crosstab shows the sum of quantities rather than just the count of transactions, revealing that CustomerID 1001 bought 2 units of A101 (from one transaction), 2 units of B205, and 1 unit of C303.

6.4.8 Advanced Visualizations: Introduction

While basic charts like bar charts and line plots are excellent for presenting simple trends and comparisons, they often fall short when dealing with complex datasets or when the goal is to uncover deeper patterns, distributions, and relationships. Advanced visualization methods provide more sophisticated ways to represent data, allowing for richer insights and a more comprehensive understanding.

These advanced techniques go beyond merely showing aggregated numbers; they help to:

- Identify correlations between variables.
- Understand the distribution of data points.
- Spot outliers or anomalies.
- Explore multi-dimensional relationships.
- Compare multiple groups or variables simultaneously.

In the following sections, we will delve into commonly used advanced visualizations using Matplotlib, a fundamental plotting library in Python.

6.4.8.1 Scatter Plots in Matplotlib

Scatter plots are a fundamental and highly effective visualization tool used to display the relationship between two continuous (numeric) variables. Each point on a scatter plot represents an observation, with its position determined by its values on the x and y axes. They are particularly useful for:

- Identifying correlations: Visually determine if there's a positive, negative, or no correlation between the variables.
- Detecting clusters: Observe if data points tend to group together.
- Spotting outliers: Identify individual points that deviate significantly from the overall pattern.

Matplotlib's plt.scatter() function is used to create scatter plots.

Key parameters of plt.scatter():

- x, y: The data positions (coordinates of the points).
- s: Marker size (can be a single value or an array).
- c: Marker color (can be a single color or an array to color-encode points based on a third variable).

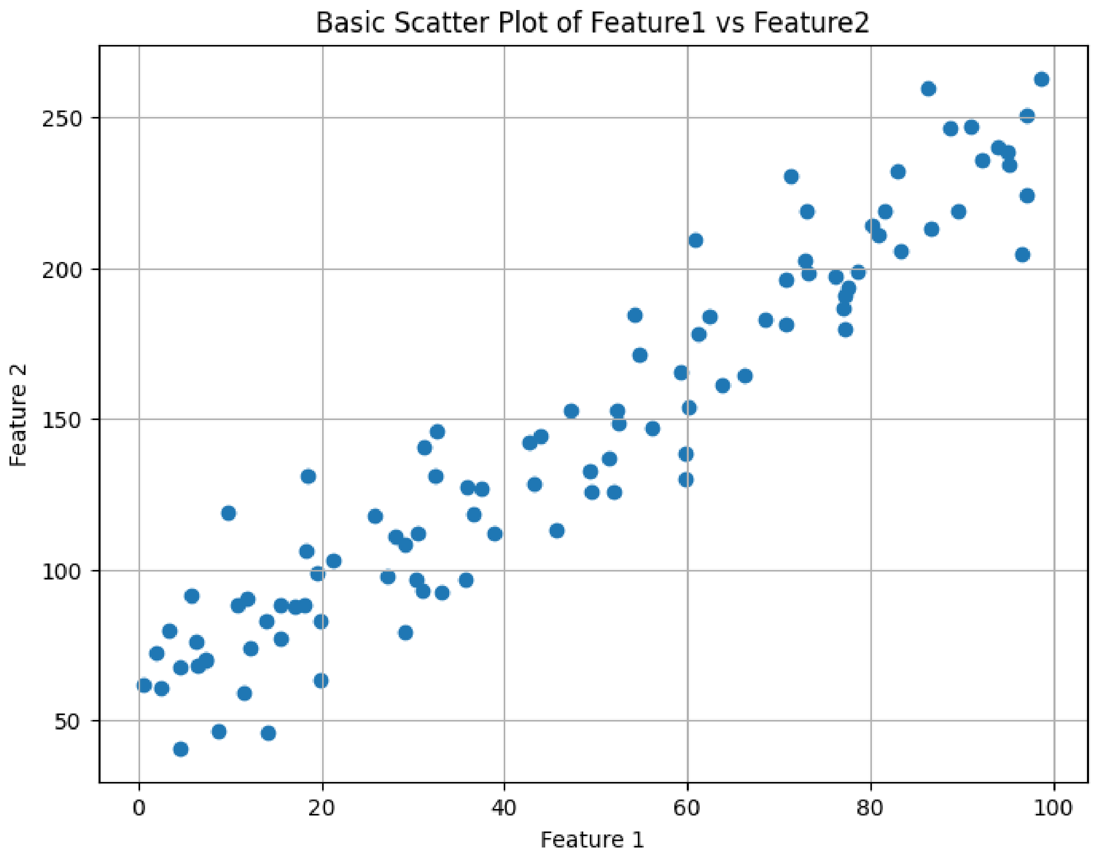
- marker: The style of the markers (e.g., ‘o’ for circle, ‘x’ for cross, ‘s’ for square).
- alpha: Transparency of the markers (0.0 fully transparent, 1.0 fully opaque).

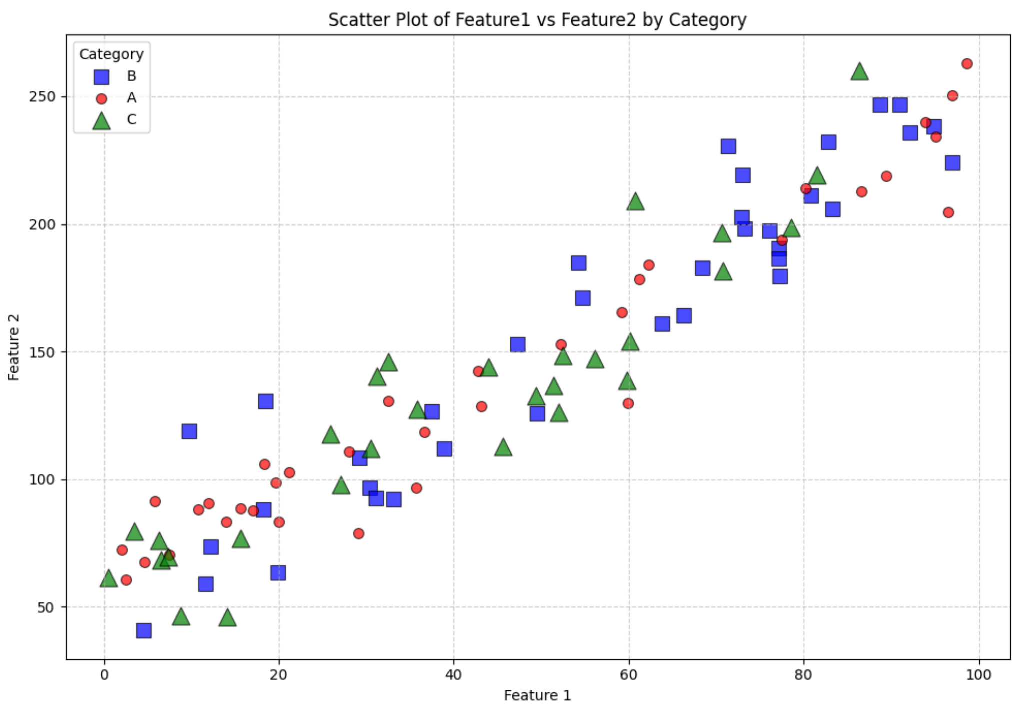
Example:

Let’s generate some sample data to visualize a relationship between Feature1 and Feature2, and then add a third variable, Category, for color encoding.

```
import matplotlib.pyplot as plt
import numpy as np
# Generate sample data
np.random.seed(42)
data_points = 100
feature1 = np.random.rand(data_points) * 100
feature2 = 2 * feature1 + np.random.randn(data_points) * 20 + 50
categories = np.random.choice([‘A’, ‘B’, ‘C’], data_points)
# Create a DataFrame for convenience
scatter_df = pd.DataFrame({
   ‘Feature1’: feature1,
   ‘Feature2’: feature2,
   ‘Category’: categories
})
# Basic Scatter Plot
plt.figure(figsize=(8, 6))
plt.scatter(scatter_df[‘Feature1’], scatter_df[‘Feature2’])
plt.title(‘Basic Scatter Plot of Feature1 vs Feature2’)
plt.xlabel(‘Feature 1’)
plt.ylabel(‘Feature 2’)
plt.grid(True)
plt.show()
# Scatter Plot with Color Encoding by Category and varying sizes
plt.figure(figsize=(10, 7))
colors = {‘A’: ‘red’, ‘B’: ‘blue’, ‘C’: ‘green’}
sizes = {‘A’: 50, ‘B’: 100, ‘C’: 150} # Varying sizes for demonstration
for category in scatter_df[‘Category’].unique():
    subset = scatter_df[scatter_df['Category'] == category]
    plt.scatter(subset['Feature1'], subset['Feature2'],
                color=colors[category],
                s=sizes[category],
                label=category,
                alpha=0.7, # Add some transparency
                edgecolors='w', # White edges for better visibility
               marker='o' if category == 'A' else ('s' if category == 'B' else '^')) # Different
markers
plt.title('Scatter Plot of Feature1 vs Feature2 by Category')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(title='Category')
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

Output :





A basic scatter plot will show a general positive linear relationship between Feature1 and Feature2, albeit with some noise.The second scatter plot will display points colored and sized according to their Category, with different marker shapes, allowing visual assessment of whether the relationship differs across categories.

For example, you might observe that Category 'A' points (red circles) tend to have lower values of Feature1 and Feature2, while Category 'C' points (green triangles) have higher values, even if the overall positive trend holds for all categories. This visual distinction helps in identifying potential sub-patterns within the data.

6.4.8.2 Histograms in Matplotlib

Histograms are powerful graphical representations used to display the distribution of a single continuous (numeric) variable. They divide the data into a series of intervals (called bins) and then count how many data points fall into each bin. The height of each bar in the histogram represents the frequency (or proportion) of data points within that bin.

Histograms are crucial for:

- Understanding data distribution: Symmetrical, skewed (left or right), bimodal, uniform.
- Identifying central tendency: Where most of the data lies.
- Detecting spread/variability: How dispersed the data is.
- Spotting outliers or unusual observations.

Matplotlib's plt.hist() function is used to create histograms.

Key parameters of plt.hist():

x: The input data (a single array or list of values).

bins: The number of bins or a sequence of bin edges.

range: The lower and upper range of the bins.

density: If True, the histogram will be normalized to form a probability density, i.e., the area under the bars sums to 1.

color: Color of the histogram bars.

edgecolor: Color of the bin edges.

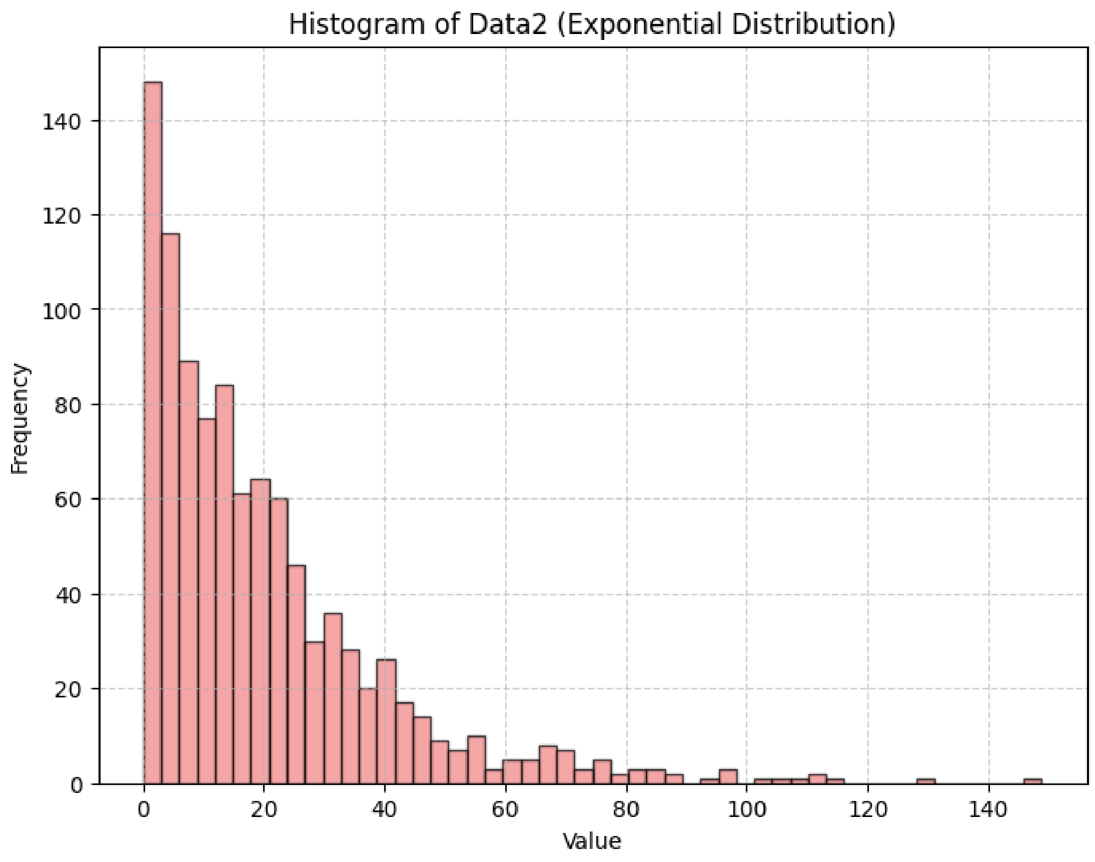
alpha: Transparency of the bars.

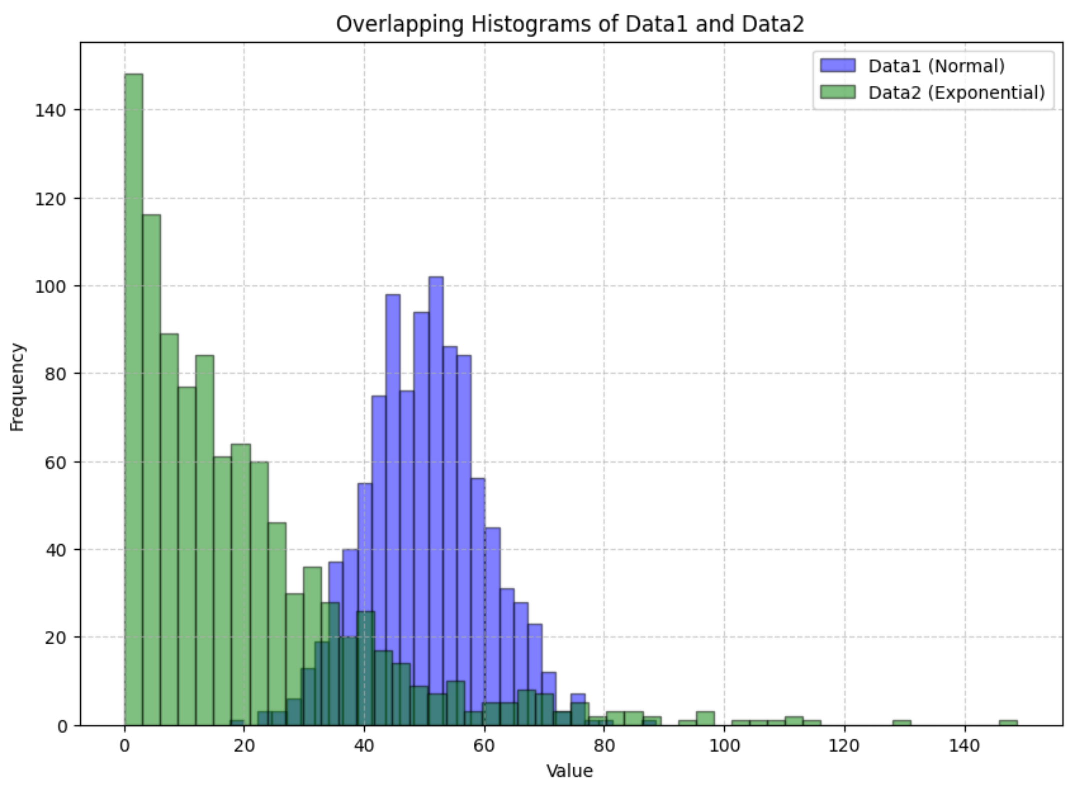
Example:

Let's generate data for two features with different distributions and visualize them using histograms.

```
# Generate sample data for histograms

np.random.seed(42)

data1 = np.random.normal(loc=50, scale=10, size=1000) # Normally distributed

data2 = np.random.exponential(scale=20, size=1000) # Exponentially distributed

# Histogram for data1 (Normal Distribution)

plt.figure(figsize=(8, 6))

plt.hist(data1, bins=30, color='skyblue', edgecolor='black', alpha=0.7)

plt.title('Histogram of Data1 (Normal Distribution)')

plt.xlabel('Value')

plt.ylabel('Frequency')

plt.grid(True, linestyle='--', alpha=0.6)

plt.show()

# Histogram for data2 (Exponential Distribution)

plt.figure(figsize=(8, 6))

plt.hist(data2, bins=50, color='lightcoral', edgecolor='black', alpha=0.7)

plt.title('Histogram of Data2 (Exponential Distribution)')

plt.xlabel('Value')

plt.ylabel('Frequency')
plt.grid(True, linestyle='--', alpha=0.6)

plt.show()

# Overlapping Histograms for comparison

plt.figure(figsize=(10, 7))

plt.hist(data1, bins=30, color='blue', edgecolor='black', alpha=0.5, label='Data1 (Normal)')

plt.hist(data2, bins=50, color='green', edgecolor='black', alpha=0.5, label='Data2 (Exponential)')

plt.title('Overlapping Histograms of Data1 and Data2')

plt.xlabel('Value')

plt.ylabel('Frequency')

plt.legend()

plt.grid(True, linestyle='--', alpha=0.6)

plt.show()
```

Output :





The first histogram will show a bell-shaped curve, typical of a normal distribution, centered around 50.The second histogram will display a skewed distribution, with a high frequency of low values and a long tail to the right, characteristic of an exponential distribution.The third plot will show both distributions overlaid, allowing for a direct visual comparison of their shapes and ranges.

Interpreting these shapes helps in understanding the underlying processes that generate the data. For instance, a normal distribution might suggest a stable process, while a skewed distribution could indicate an asymmetry, such as income distribution where most people earn less and a few earn significantly more.

6.4.8.3 Creating Subplots

When presenting multiple related visualizations, displaying them in a single figure often enhances comparability and narrative flow. Subplots allow you to arrange multiple plots within a single figure. Matplotlib's plt.subplots() function is the recommended way to create a figure and a set of subplots.

plt.subplots() returns two objects:

- fig: The Figure object, which is the top-level container for all plot elements.
- ax (or axes): The Axes object(s), which is the actual plot area where data is drawn. If creating multiple subplots, ax will be an array of Axes objects.

Key parameters of plt.subplots():

- nrows: Number of rows in the subplot grid.

- ncols: Number of columns in the subplot grid.
- figsize: A tuple (width, height) in inches for the figure size.
- sharex, sharey: Boolean or ‘all’, ‘row’, ‘col’ to share x- or y-axis properties among subplots.

Example:

Let’s create a figure with two subplots: a scatter plot on one side and a histogram on the other, using our previously generated data.

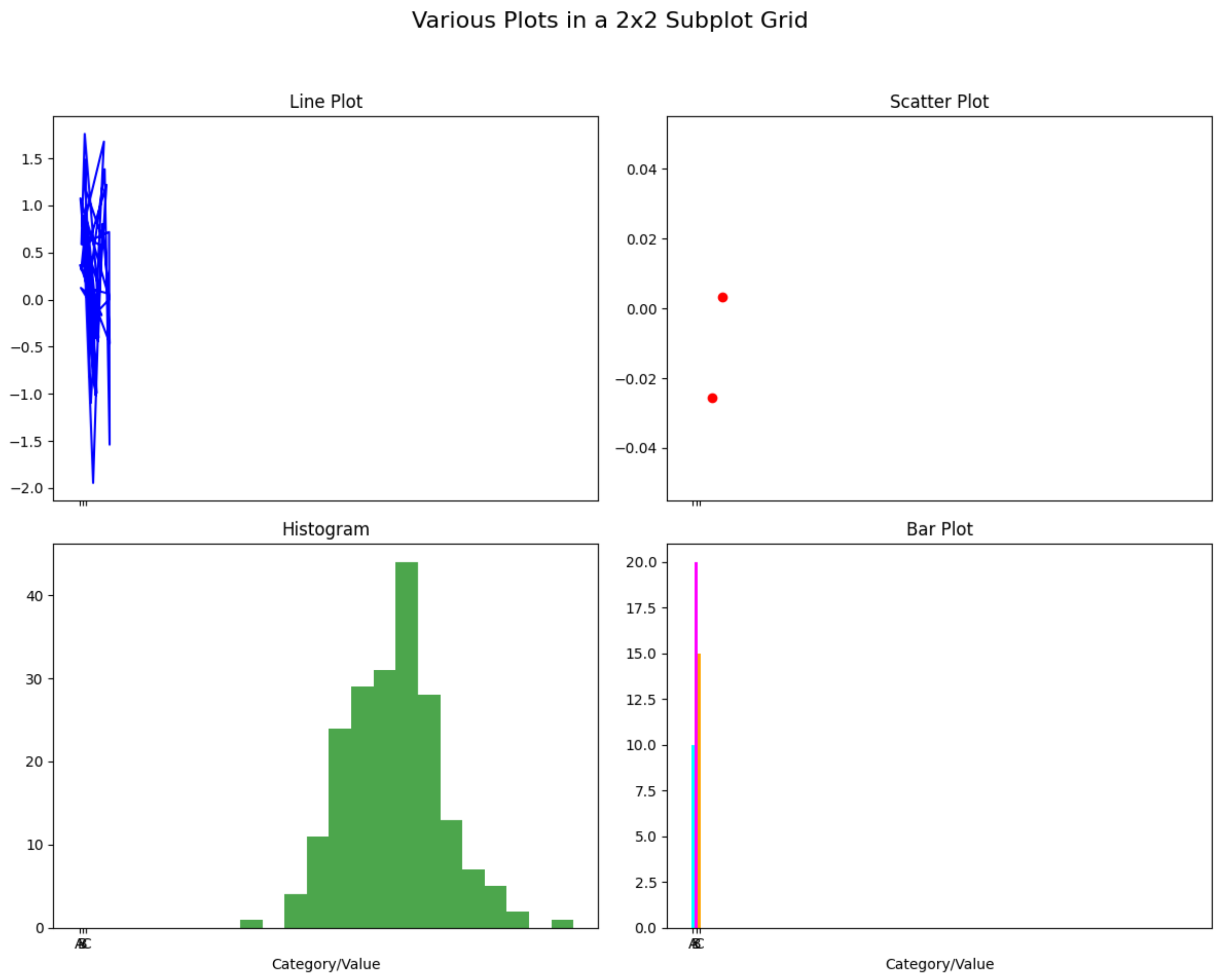
```
# Create sample data for subplots

np.random.seed(42)

x_data = np.random.rand(50) * 10

y_data = np.sin(x_data) + np.random.randn(50) * 0.5

hist_data = np.random.normal(loc=100, scale=15, size=200)

# Create a figure with 1 row and 2 columns of subplots

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14, 6))

# Plot 1: Scatter plot on the first Axes object (axes[0])

axes[0].scatter(x_data, y_data, color=’purple’, alpha=0.7)

axes[0].set_title(‘Scatter Plot of X vs Y’)

axes[0].set_xlabel(‘X Value’)

axes[0].set_ylabel(‘Y Value’)

axes[0].grid(True, linestyle=’:’, alpha=0.6)

# Plot 2: Histogram on the second Axes object (axes[1])

axes[1].hist(hist_data, bins=20, color=’teal’, edgecolor=’black’, alpha=0.8)

axes[1].set_title(‘Histogram of Sample Data’)

axes[1].set_xlabel(‘Value’)

axes[1].set_ylabel(‘Frequency’)

axes[1].grid(True, linestyle=’:’, alpha=0.6)

# Adjust layout to prevent overlapping titles/labels

plt.tight_layout()

plt.show()
```

```
# Example with a 2x2 grid, sharing x-axis

fig2, axes2 = plt.subplots(nrows=2, ncols=2, figsize=(12, 10), sharex=True)

# Flatten the axes array for easier iteration

axes2 = axes2.flatten()

# Plotting different types of plots in each subplot

axes2[0].plot(x_data, y_data, color='blue')

axes2[0].set_title('Line Plot')

axes2[1].scatter(x_data, y_data, color='red')

axes2[1].set_title('Scatter Plot')

axes2[2].hist(hist_data, bins=15, color='green', alpha=0.7)

axes2[2].set_title('Histogram')

axes2[3].bar(['A', 'B', 'C'], [10, 20, 15], color=['cyan', 'magenta', 'orange'])

axes2[3].set_title('Bar Plot')

# Set common X label for the bottom row

for i in [2, 3]:

    axes2[i].set_xlabel('Category/Value')

plt.suptitle('Various Plots in a 2x2 Subplot Grid', fontsize=16) # Super title for the entire figure

plt.tight_layout(rect=[0, 0.03, 1, 0.95]) # Adjust layout to make space for suptitle

plt.show()
```

Output:

The first figure will display two plots side-by-side: a scatter plot showing a sinusoidal relationship with noise, and a histogram illustrating a normal distribution.The second figure will show a 2x2 grid of different plot types (line, scatter, histogram, bar), demonstrating how diverse visualizations can be combined and managed within a single figure using shared axes for better comparison.

Subplots are invaluable for comparative analysis, allowing viewers to easily spot trends or differences across different data views without flipping between multiple individual plots.


Various Plots in a 2x2 Subplot Grid
Line Plot
Scatter Plot
Histogram
Bar Plot
Category/Value
Category/Value
ABC
ABC

Recap

- Data aggregation is the process of summarizing raw data into a more concise, meaningful form.
- It's crucial for transforming large datasets into actionable insights and identifying trends.
- The groupby() method in Pandas is central to data grouping, splitting a DataFrame based on one or more criteria.
- The groupby() process follows three steps: Splitting, Applying a function, and Combining the results.
- Common Pandas aggregation functions include sum(), mean(), count(), min(), and max().
- A groupby() operation returns a "lazy" GroupBy object, not a DataFrame, with computations performed only when an aggregation function is called.
- The GroupBy object allows for chaining operations to write concise code, such as df.groupby('ProductID')['Quantity'].sum().
- The .agg() method enables you to apply multiple aggregation functions to one or more columns simultaneously.
- Using a dictionary with .agg() is often preferred as it allows for custom, readable names for the output columns.
- pd.pivot_table() is a powerful function for summarizing and reorganizing data into a multi-dimensional table.
- Key parameters of pivot_table() include values, index, columns, and aggfunc.
- The margins=True parameter in a pivot table adds a grand total for rows and columns.
- pd.crosstab() is a specialized function used for cross-tabulation, which counts the frequency of combinations between categorical variables.
- Scatter plots are a fundamental visualization for displaying the relationship between two continuous variables.
- Scatter plots are useful for identifying correlations, detecting clusters, and spotting outliers.

Objective Type Questions

1. Which Pandas method groups data?
2. What kind of object does groupby() return?
3. What function calculates a group's total?
4. Which method applies many aggregations?
5. What summarizes data into a reshaped table?
6. Which function creates frequency tables?
7. What plot shows relationships between two numbers?
8. What displays a variable's distribution?
9. What Matplotlib function makes multiple plots in one figure?
10. What is the term for improving visuals?

Answers to Objective Type Questions

1. groupby
2. GroupBy
3. sum
4. agg
5. PivotTable
6. crosstab
7. Scatter
8. Histogram
9. subplots
10. Customization

Assignments

For each question, write a Python script using pandas and matplotlib to solve the problem. You can use the provided sample DataFrame for questions 1-6.

```
import pandas as pd

import numpy as np

# Sample DataFrame for Questions 1-6

data = {

    ‘OrderID’: [101, 102, 103, 104, 105, 106, 107, 108],

    ‘CustomerID’: [1001, 1002, 1001, 1003, 1002, 1004, 1003, 1001],

    ‘ProductID’: [‘A101’, ‘B205’, ‘C303’, ‘A101’, ‘B205’, ‘A101’, ‘C303’,
‘B205’],

    ‘Category’: [‘Electronics’, ‘Books’, ‘Electronics’, ‘Electronics’, ‘Books’,
‘Electronics’, ‘Electronics’, ‘Books’],

    ‘Quantity’: [2, 1, 1, 3, 1, 2, 1, 4],

    ‘UnitPrice’: [150.00, 25.00, 50.00, 150.00, 25.00, 150.00, 50.00, 25.00],

   ‘PurchaseDate’: [‘2023-08-01’, ‘2023-08-01’, ‘2023-08-02’, ‘2023-08-02’,
‘2023-08-03’, ‘2023-08-03’, ‘2023-08-04’, ‘2023-08-04’]

}

df = pd.DataFrame(data)

df[‘PurchaseDate’] = pd.to_datetime(df[‘PurchaseDate’])

df[‘TotalPrice’] = df[‘Quantity’] * df[‘UnitPrice’]
```

1. Basic Grouping and Aggregation: Using the sample DataFrame, find the total quantity sold for each Category. Print the result.

2. Multiple Aggregations: Using the groupby() and .agg() methods, calculate the sum of TotalPrice and the mean of Quantity for each ProductID. Rename the resulting columns to ‘TotalRevenue’ and ‘AverageQuantity’ respectively. Print the final DataFrame.

3. Creating a Pivot Table: Create a pivot table to show the total Quantity sold for each ProductID on each PurchaseDate. Fill any missing values with 0.

4. Pivot Table with Margins: Recreate the pivot table from Question 3, but this time include row and column totals by using the margins=True parameter. Name the totals 'Grand Total'.

5. Basic Cross-tabulation: Use pd.crosstab() to create a frequency table that shows the number of transactions for each combination of CustomerID and Category.

6. Cross-tabulation with Aggregation: Use pd.crosstab() to create a table that shows the sum of the TotalPrice for each combination of CustomerID and Category. Use fill_value=0 to handle missing combinations.

7. Basic Scatter Plot: Generate a DataFrame with two columns of random numbers, x and y, each with 100 data points. Create and display a basic scatter plot to visualize the relationship between x and y. Make sure to add a title and axis labels.

8. Advanced Scatter Plot: Create a new DataFrame with three columns: Size, Weight, and Type (where Type is a categorical variable, e.g., 'Small', 'Medium', 'Large'). Generate 150 random data points for each column. Create a scatter plot where the marker size is determined by the Size column, and the color of the points is determined by the Type column. Include a legend for the Type variable.

Reference

1. Hunter, J. D. (2007). *Matplotlib: A 2D graphics environment*. *Computing in Science & Engineering*, 9(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

2. VanderPlas, J. (2016). *Python Data Science Handbook*. O'Reilly Media.

3. McKinney, W. (2017). *Python for Data Analysis* (2nd ed.). O'Reilly Media.

4. Matplotlib Developers. (2024). *Matplotlib Documentation*. Retrieved from https://matplotlib.org/stable/

Suggested Reading

1. Swaroop, C. H. (2019). *Python Programming: A Modern Approach*. Oxford University Press.
2. Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.
3. Ascher, D., & Lutz, M. (2003). *Learning Python* (2nd ed.). O'Reilly Media.
4. Yadav, T. (2021). *Data Science with Python*. BPB Publications.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ……… Reg. No : ....

Name : ……...........

Model Question Paper- set-I

End Semester Examination

BSc. Data Science and Analytics

B24DS05DC:DATA STRUCTURES

(CBCS - UG)

2024-25 - Admission Onwards

Time: 2 Hours Max Marks: 45

Section A

Answer any 10 questions. Each carries one mark **(10 x 1 = 10)**

1. What keyword is used to define a constant in C?
2. Which of the following is a non-primitive data structure?
   a. Char
   b. Float
   c. Queue
   d. Boolean
3. Name one application of a linked list.
4. A binary tree in which every node has 0 or 2 children?
5. What is heap?
6. Write the classifications of non-premitive data types.
7. Which statement allows multi-way branching?
8. What is the index of the first element in an array?
9. Name two basic operations performed on a linked list.

10. What is the main data structure used in hashing?

11. What is trie data structure?

12. List different types of Queue.

13. What keyword is used to define a structure in C?

14. Which collision method uses a second hash function?

15. Which type of linked list has its last node pointing back to the first node?

Section B

Answer any 5 questions. Each carries two marks **(5 x 2= 10)**

16. What is Dynamic Memory Allocation ?

17. What is a one-dimensional array?

18. What is meant by traversal in a linked list?

19. What is an undirected graph?

20. Explain any two Properties of Hash Functions.

21. Explain the advantages of Red Black Tree?

22. What is a pointer to an array?

23. Explain any two applications of stack in briefly.

24. What is a self-referential structure?

25. What is the degree of a vertex in a graph?

Section C

Answer any 5 questions. Each carries four marks **(5 x 4 = 20)**

26. Explain the different representations of heap.

27. Explain the process of balancing a Red-Black Tree after deletion with a suitable case.

28. Differentiate between entry-controlled and exit-controlled loops.

29. Define a linear data structure with an example.

30. Explain the representation and basic operations of a stack using a linked list.

31. Draw a binary tree of height 3 and label root, internal, and leaf nodes.

32. Distinguish between min-priority queue and max-priority queue

33. Describe a circular queue.

34. Explain any 4 operators used in C

35. Explain any two primary operations of a stack.

Section D

Answer any 2 questions. Each carries fifteen mark **(2 x 15 = 30)**

36. Explain the heap data structure in detail ? Differentiate between min-heap and max-heap with suitable examples and diagrams.

37. Write a C program to define a structure called Student with members: roll_no, name, and marks. Create an array of 5 students, accept details from the user, and display the student with the highest marks.

38. Explain the Queue data structure in detail. Include its basic operations, and explain how the FIFO principle works with an example.

39. Explain different collision resolution techniques in hashing and explain its advantages and disadvantages.

SREENARAYANAGURU OPEN UNIVERSITY

QP CODE: ………　　　　　　　　　　Reg. No : ....

Name : ……...........

Model Question Paper- set-I

End Semester Examination

BSc. Data Science and Analytics

B24DS05DC:DATA STRUCTURES

(CBCS - UG)

2024-25 - Admission Onwards

Time: 2 Hours　　　　　　　　　　Max Marks: 45

Section A

Answer any 10 questions. Each carries one mark **(10 x 1 = 10)**

1. The expression a ? b : c is an example of which operator?
2. What does LIFO stand for?
3. Name the data structure used to represent a queue using a linked list.
4. Binary tree filled from left to right is?
5. What is Hashing?
6. What is the use of peek operation in stack?
7. In a doubly linked list, which pointer points to the previous node?
8. What type of queue allows insertion and deletion from both ends?
9. What is the pointer called that refers to the first node in a linked list?
10. Traversal visiting nodes in Left-Root-Right order?
11. Which collision resolution technique uses a linked list at each index of a hash table?
12. Which type of data structure is Queue?

13. In a linked stack, which operation adds an element?

14. What is the total number of elements in a 2D array declared as int arr[4][5];?

15. Identify the data structure where insertion happens at one end and deletion happens at the other.

Section B

Answer any 5 questions. Each carries two marks **(5 x 2= 10)**

16. What are Nested loops?

17. Differentiate between array and linked list.

18. Name two types of linked lists and state one feature of each.

19. Define a tree and explain its basic terminology.

20. Write any two applications of trie.

21. Write about enqueue and dequeue operations.

22. Mention any two advantages of functions.

23. What is the difference between static and dynamic memory allocation?

24. Mention two advantages of using a circular linked list.

25. What is an AVL tree?

Section C

Answer any 5 questions. Each carries four marks **(5 x 4 = 20)**

26. Explain advantages and disadvantages of heap.

27. Write about any two advantages and disadvantages of heap.

28. Differentiate Static and Dynamic memory.

29. Describe the Tower of Hanoi problem and explain how stacks can be used to solve it.

30. Define B-tree. Explain its basic properties.

31. Define a graph and explain types of graphs with examples.

32. Explain the different methods used to represent priority queues.

33. What is a Circular linked list? Explain the basic operations of a circular linked list.

34. Why break,continue and goto statements are used?

35. Write a short note on application of queues.

Section D

Answer any 2 questions. Each carries fifteen mark **(2 x 15 = 30)**

36. Draw a binary tree of height 4 and show all traversals.

37. Discuss advantages of BST over a linear data structure like linked list.

38. Explain basic and advanced data structures, and compare their differences.

39. Explain about the implementation of priority queues using array, linked list and heap.

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

DON'T LET IT BE TOO LATE

SAY NO TO DRUGS

LOVE YOURSELF AND ALWAYS BE HEALTHY

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

Introduction to Python Programming

COURSE CODE: B24DS06DC

Sreenarayanaguru Open University

Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841